Estudio y aplicación de metaheurísticas y comparación con métodos exhaustivos

Realizado por:

Jhonny Vargas Paredes
Víctor Penit Granado

Trabajo de Fin de Grado del Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

Curso 2015/2016

Director:
Pablo M. Rabanal Basalo

Departamento de Sistemas Informáticos y Computación

Autorización de utilización y difusión

Los abajo firmantes, alumno/s y tutor/es del Trabajo Fin de Grado (TFG) en el Grado en Ingeniería del Software de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el Trabajo Fin de Grado (TFG) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Título del TFG: "Estudio y aplicación de metaheurísticas y comparación con métodos exhaustivos"

Curso académico: 2015/2016

Nombre del Alumno/s:
Jhonny Vargas Paredes
Víctor Penit Granado

Tutor/es del TFG y departamento al que pertenece:
Pablo M. Rabanal Basalo
Departamento de Sistemas Informáticos y Computación

Dedicatorias

Jhonny Vargas Paredes

En especial, a mi querida madre, que un día tuvo que partir de nuestra tierra en busca de un futuro mejor para nuestra familia. Gracias por inculcarme esos grandes valores de trabajo y disciplina desde que era un niño, gracias por tus innumerables atenciones, por tu apoyo durante todos estos años de estudio. En definitiva, fundamentalmente soy quien soy gracias a ti.

Un lugar muy importante para ti tía Margarita, que hoy ya no estas entre nosotros, pero dejaste huella en mí y me enseñaste a luchar contra las adversidades del camino. Siempre estarás presente en cada paso de mi vida.

A todos aquellos profesores que durante esta etapa supieron transmitirme los preciados conocimientos que ahora poseo.

A esos grandes compañeros de estudio, con los que hemos compartido las alegrías y tristezas propias de esta etapa que ya termina. Son y serán siempre grandes amigos.

Al resto de personas que están y estuvieron en mi vida, y que de una u otra manera me ayudaron a crecer como persona.

Finalmente, a todos los estudiantes de ingeniería, porque con su esfuerzo y dedicación siempre ponen un granito de arena más para intentar que este mundo algún día sea mejor.

Víctor Penit Granado

Muchas gracias a mi familia, amigos y a mi novia, que tanto me han apoyado y animado en este curso.

"Los dos días más importantes de tu vida son el día en que naces y el día en que descubres por qué", Mark Twain.

Resumen

Cuando nos enfrentamos a problemas reales haciendo uso de recursos computacionales, hemos de tener en cuenta que el número de posibles soluciones candidatas a tener en cuenta puede llegar a ser tan inmenso que abordarlas mediante técnicas algorítmicas clásicas, en la mayoría de los casos, pueden llegar a convertirse en un problema en sí mismo debido al gran coste en recursos que pueden llegar a generar. En este contexto, aspectos como el tiempo utilizado en la búsqueda de una solución mediante algoritmos de búsqueda exhaustiva tales como fuerza bruta, vuelta atrás, ramificación y poda, etc., puede llegar a ser prohibitivo en la práctica. Ante este problema que se nos plantea, podemos hacer un estudio sobre otros métodos, tales como los *metaheurísticos*, que, aunque no siempre aseguran la *optimalidad* de las soluciones producidas; tienen un tiempo de ejecución mucho menor que los métodos exhaustivos.

En el presente trabajo hemos seleccionado dos problemas NP-completos de entre los más famosos de la literatura y hemos realizado un estudio de ambos. Concretamente, los problemas seleccionados han sido el *TSP (Traveling Salesman Problem)* y el *problema de la Mochila 0-1*. Por otro lado, hemos llevado a cabo un estudio sobre distintas metaheurísticas para poder resolver los problemas mencionados. Entre estas metaheurísticas, hemos seleccionado cuatro: *metaheurísticas evolutivas*, *metaheurísticas inspiradas en colonias de hormigas*, *metaheurísticas simulated annealing (enfriamiento simulado)* y *metaheurísticas GRASP (Greedy Randomized Adaptive Search Procedure)*.

Después de esto, cada problema ha sido resuelto aplicando tanto *algoritmos de búsqueda exhaustiva* como metaheurísticas. Una vez adaptados los algoritmos a la resolución de los problemas concretos, hemos realizado un estudio experimental, donde se realizaron comparativas de rendimiento.

Finalmente, todo este trabajo ha sido plasmado en el desarrollo de una aplicación software, la cual consta de dos partes: una que contiene la implementación los algoritmos adaptados para la resolución de los problemas y que son ofrecidos a modo de servicios web y otra parte donde se ha implementado un cliente web que puede consumir estos servicios y realizar una presentación más vistosa de la ejecución de los algoritmos y los resultados obtenidos. Esta arquitectura podrá servir como base para futuras ampliaciones de este estudio.

Abstract

When we face real problems using computational resources, we must take into account that the number of possible candidate solutions to consider can become so huge that address this through classic algorithmic techniques, in most of cases can become a problem itself due to the high cost in of generated resources. In this context, aspects such as the time spent in the search for a solution by exhaustive search algorithms such as backtracking, branch and bound, dynamic programming, etc., can become prohibitive in practice. To face this problem, we can make a study of other methods, such as *metaheuristics*, which, although it not always ensure the optimality of the solutions produced, have a runtime much smaller than the exhaustive methods.

In this work we have selected two NP-complete problems among the most famous in literature, and we have studied both. Specifically, the selected problems were the *TSP (Traveling Salesman Problem)* and *0-1 Knapsack problem*. On the other hand, we have carried out a study on different metaheuristics to solve the above problems. We have selected four of these techniques: *evolutionary metaheuristics*, *metaheuristics inspired in ant colonies*, *simulated annealing metaheuristics* and *GRASP (Greedy Randomized Adaptive Search Procedure) metaheuristics*.

After this, each problem has been solved by applying exhaustive search algorithms and metaheuristics. Once the algorithms have been adapted to solving the specific problems, we have performed an experimental study where we have made comparatives of performance.

Finally, all this work has led to the development of a software application, which is composed of two parts: one containing the implementation of the algorithms adapted to solve the problems offered as web services, and another part where a web client has been implemented to consume these services, and present in a more attractive manner the results obtained by the algorithms. This architecture will serve as a basis for future extensions of this study.

Palabras clave

- Problemas NP-completos
- Metaheurística
- Problema del viajante de comercio
- Problema de la mochila
- Optimización
- Optimización de colonia de hormigas
- Algoritmos genéticos
- Enfriamiento simulado
- Procedimiento de búsqueda adaptativa aleatorizado voraz

Keywords

- NP-complete problems
- Metaheuristic
- Traveling salesman problem
- Knapsack problem
- Optimization
- Ant colony optimization
- Genetic algorithms
- Simulated annealing
- Greedy randomized adaptive search procedure

Índice

Índice de ilustraciones

Índice de tablas

"La informática tiene que ver con los ordenadores lo mismo que la astronomía con los telescopios", Edsger W. Dijkstra.

Capítulo 1

Introducción

En este capítulo hablamos sobre la motivación que nos ha llevado a realizar este Trabajo de Fin de Grado, resumimos los principales objetivos que perseguimos, y cómo hemos estructurado esta memoria.

1.1. Motivación

La vida misma está llena de decisiones, y ya sean éstas simples o complejas, surgen como respuesta a los problemas que se nos plantean y nos decantamos por unas u otras atendiendo a ciertos criterios. Este proceso de toma de decisiones es una labor extremadamente compleja, puesto que, aunque tengamos todas las opciones de decisión disponibles, es difícil elegir entre todas para salir airosos. Para llevar a cabo esta elección tenemos que atender a ciertos criterios, para ver cuál es la decisión que más nos conviene.

En ciertas situaciones, podemos intuir decisiones de manera muy rápida que suelen llevarnos hacia buenas soluciones para el problema que se nos plantea, aunque quizá no sean las mejores. Pero para tomar las mejores decisiones, tendríamos que establecer un método de búsqueda y valoración de todas las opciones posibles, lo cual sería muy laborioso para el ser humano.

Gracias a los avances en las ciencias de la computación, podemos efectuar rápidamente grandes procesos computacionales compuestos de múltiples operaciones, las cuales un ser humano tardaría en resolverlas un tiempo demasiado grande. Sin embargo, los procesadores, que según pasa el tiempo son cada vez más potentes, pueden llevar a cabo este enorme conjunto de operaciones en cuestión de segundos.

Aun así y a pesar de la gran velocidad de los computadores actuales, existen determinados tipos de problemas, para los cuales alcanzar una solución óptima, se requiere de un método que consiste en ejecutar un número demasiado grande operaciones, provocando que el proceso requerido tarde un tiempo exagerado, llegando incluso a superar el tiempo de existencia del universo.

Para acelerar los procesos resolutivos de los problemas, podemos mejorar tanto el hardware como el software. Nosotros nos centraremos en el software, en los algoritmos.

Un algoritmo es un conjunto de pasos para resolver un problema. Existen muchos tipos de algoritmos, y a medida que se hacen más estudios, se encuentran algoritmos más eficientes. Muchos algoritmos son intuitivos, otros son inspirados en la naturaleza, otros son elaboraciones

más complejas, por ejemplo, basados en estudios estadísticos o matemáticos.

Lo que pretendemos en este Trabajo de Fin de Grado es estudiar una serie de algoritmos y comparar su eficiencia, según puedan ser adaptados de manera más o menos flexible a unos determinados problemas.

Es por todas estas razones, que la realización de este trabajo nos ha parecido un buen reto y una gran oportunidad para aprender todo lo que podamos sobre este interesante campo de investigación.

1.2. Objetivos

El objetivo principal de este trabajo es el *estudio de diferentes métodos metaheurísticos*, los cuales compararemos al implementarlos para la resolución de una serie de problemas NP-completos. Para poder realizar la comparación, además realizaremos un estudio de estos problemas. Concretamente elegiremos aquellos que puedan tener aplicaciones prácticas más evidentes en el mundo real, y los modelizaremos para poder resolverlos. Y para poder optimizar, y comparar, para en algunos casos saber la solución óptima, *haremos un estudio de algunas heurísticas y algoritmos exactos*, que puedan resolver los problemas NP-completos elegidos. Una vez hechas las comparaciones, procederemos a un estudio por medio de tablas y gráficas, con las que sacaremos nuestras propias conclusiones.

Además, como estudiantes de Ingeniería del Software, para llevar a cabo este estudio, crearemos una aplicación que nos ofrezca los resultados que necesitaremos, de manera que en el transcurso del desarrollo de dicha aplicación, llevaremos a cabo técnicas de Ingeniería del Software para poder llevar una gestión correcta del proyecto, y utilizaremos patrones de programación para desarrollar una aplicación que resulte fácilmente mantenible y escalable, con el fin de poder ampliar en un futuro este trabajo.

Teniendo en cuenta que a lo largo del plan de estudios de la carrera no vimos estos métodos algorítmicos, nos parece esta una buena oportunidad para adentrarnos en materia y aprender todo lo que sea posible, mientras desarrollamos este Trabajo de Fin de Grado.

1.3. Estructura del documento

A continuación, pasamos a explicar el propósito los capítulos que componen esta memoria, a partir del capítulo 2:

- El capítulo 2 está dedicado a la presentación de diferentes conceptos teóricos sobre optimización, complejidad, etc., que consideramos básicos para hacer entrar al lector en materia.

- En el capítulo 3 presentamos los problemas NP-completos elegidos y que hemos llegado a modelizar y resolver. Se hace una breve reseña histórica para cada uno, se explica en qué consisten y se hace énfasis en su modelización matemática.

- En el capítulo 4 se explican tanto los algoritmos exactos como las heurísticas y metaheurísticas estudiadas y que hemos implementado para resolver los problemas elegidos. Se explica en qué consiste su funcionamiento, y se proporciona un esquema general de estos.

- El capítulo 5 contiene información relativa a la gestión del proyecto. En él se incluye un plan de trabajo y una planificación temporal.

- En el capítulo 6 explicamos cómo hemos adaptado las implementaciones de los algoritmos exactos, las heurísticas y las metaheurísticas para la resolución de los problemas elegidos. También explicamos las estructuras y tipos de datos que hemos construido para representar y organizar la información de los problemas dentro de la aplicación.

- El capítulo 7 muestra para cada problema, un estudio comparativo entre los algoritmos aplicados, siguiendo diferentes criterios.

- El capítulo 8 cuenta cómo hemos desarrollado la aplicación destinada a este estudio, y qué tecnologías hemos utilizado para ello. Además, al hablar sobre la arquitectura, explicamos cada una de las capas que la componen. Además, presentamos un manual de uso de la aplicación, acompañado de imágenes.

- En el capítulo 9 exponemos las conclusiones personales que hemos sacado al finalizar este Trabajo de Fin de Grado.

"Frecuentemente hay más que aprender de las preguntas inesperadas de un niño que de los discursos de un hombre", John Locke.

Capítulo 2

Conceptos

Con este capítulo pretendemos presentar algunas bases teóricas que hemos considerado importantes y que surgen cuando tratamos de resolver problemas aplicando metaheurísticas o métodos exhaustivos. Se expondrán conceptos importantes tales como optimización, complejidad algorítmica, problemas P, problemas NP, etc.

2.1. Investigación operativa y resolución de problemas

La investigación operativa es la rama de las matemáticas encargada de analizar los problemas resultantes de diferentes aspectos de la vida real y de tomar decisiones sobre ellos. Típicamente busca optimizar un aspecto cuantificable mediante la toma de decisiones dentro de un conjunto amplio y complejo de las mismas. Para ello se vale de métodos científicos y se sigue una serie de pasos: análisis del problema, identificación de sus elementos, modelado matemático, propuesta de método o métodos de resolución y análisis de las soluciones obtenidas.

Problemas que aborda la investigación operativa pueden ser, por ejemplo, la decisión respecto a los recorridos a tomar sobre una red de transporte público de una ciudad, o, por ejemplo, la decisión a cerca de la ubicación de servidores en internet.

Un problema se puede definir como una *función objetivo de optimización* y un conjunto de *variables de decisión*, *parámetros* y *restricciones*.

- La *función objetivo* o *función de evaluación*, es aquella que asocia a cada solución (factible), un valor que determina la calidad de esta.

- Las *variables de decisión* son las incógnitas del problema, cuyo valor compone la solución obtenida.

- Las *restricciones* son relaciones que debe cumplir o verificar la solución.

- Los *parámetros* son los datos de entrada del problema. Estos y las variables de decisión, componen la función objetivo y las restricciones.

Soluciones

Para hallar una solución, la investigación operativa generalmente representa los problemas según un modelo matemático. Una *solución* es un dato o conjunto de datos que resuelven un problema. Es el conjunto de valores que toman las variables de decisión. Tenemos estos tipos de solución:

- Una *solución factible* es una solución que satisface un problema y sus restricciones.

- Una *solución óptima* es una solución factible que maximiza o minimiza esa función de optimización. Dentro de las soluciones óptimas podemos encontrar óptimos locales y globales:

    - El *óptimo global* es el óptimo en todo el espacio de soluciones.

    - Un *óptimo local* es un punto óptimo dentro de una región del espacio de soluciones. A su vez un óptimo local puede ser un óptimo global.

Según la representación de la solución, tanto la función objetivo como las restricciones variarán, es decir, esta influye en la modelización matemática del problema y en el tamaño del espacio de soluciones.

2.2. Optimización

El *proceso de optimización* en la resolución de problemas consiste en mejorar alguna solución relativamente buena conseguida previamente o incluso llegar a conseguir una solución óptima.

La mayoría de las veces, va a ser posible conseguir una solución óptima de un problema, sin embargo, para llegar a ella será necesario utilizar métodos con una complejidad algorítmica demasiado alta, por lo que no nos conviene seguir este camino. Para ello, recurrimos a otros métodos más rápidos (se verán más adelante) con los que poder conseguir una solución lo suficientemente buena, aunque no necesariamente la óptima.

A partir de esta solución conseguida, a base de transformaciones o combinaciones, vamos creando nuevas soluciones intentando encontrar una solución mejor o incluso alcanzar la óptima. Este proceso de mejora de una solución inicial es lo que se llama *optimización*.

Optimización combinatoria

Los problemas combinatorios son aquellos en los que se maneja una cantidad finita de posibilidades generadas a partir de un número determinado de elementos. Ahora bien, si lo que deseamos es conocer la mejor manera de dar solución a un problema en vez de hacer frente a todas las posibles soluciones; estaríamos entrando en el mundo de la optimización combinatoria.

Todos los problemas de optimización combinatoria tienen en común los siguientes puntos:

- Tener un conjunto finito de soluciones implica que las variables sean discretas.

- En determinados casos es difícil encontrar una solución.

- Aunque el conjunto de soluciones candidatas sea finito, este puede llegar a ser tan inmensamente grande, que encontrar una solución se convierta en una dura tarea.

Según el sociólogo Harold Garfinkel, los problemas de optimización combinatoria *"contienen los dos elementos que hacen atractivo un problema a los matemáticos: planteamiento sencillo y dificultad de resolución."*.

Como problemas típicos tenemos: El problema de la mochila, el problema del viajante de comercio o TSP (Traveling Salesman Problem) por sus siglas en inglés (de los que hablaremos más adelante), el problema de coloración mínima, entre otros.

Como bien se dijo antes, el conjunto de soluciones candidatas a considerar, aunque sea finito, puede llegar a ser inmenso y por tanto el tiempo requerido para encontrar la solución óptima a un problema puede llegar a ser intratable. El tamaño de este conjunto de soluciones candidatas estará ligado al tamaño de los datos de entrada del problema. Así, para el TSP, se pueden experimentar aumentos en el número de posibilidades a evaluar, por ejemplo, con 4 ciudades el número de posibilidades a evaluar serían $4! = 24$, mientras que, con 20 ciudades, este número de posibilidades aumentaría desorbitadamente hasta $20! = 2432902008176640000$.

2.3. Complejidad algorítmica

La *complejidad* es una métrica para medir la eficiencia de un algoritmo, valorada generalmente en el peor caso.

La *complejidad en tiempo* de un algoritmo es el tiempo que tarda en el peor caso.
La *complejidad en espacio* de un algoritmo es el espacio que ocupa o usa durante su ejecución en el peor caso.

Se pueden realizar comparativas de eficiencia entre algoritmos, teniendo únicamente en cuenta el tamaño de los datos de entrada, independientemente de los lenguajes de programación en que están implementados, las máquinas donde son ejecutados, y del valor de sus parámetros de entrada**[1]**. La eficiencia se expresa mediante una función matemática que dependerá solo del tamaño de entrada. Tenemos los siguientes órdenes de complejidad, según la eficiencia:

- Constante: O(1).
- Lineal: O(n).
- Logarítmica: O(log n).
- Cuadrada: O(x^2), Cúbica O(x^3), …, de orden k O(x^k).
- Combinación en producto de distintos tipos de complejidades.
  Por ejemplo, nlog = O(n) * O(logn).
- Exponencial: O(k^n). Por ejemplo: O(2^n), O(3^n), …
- Factorial: O(n!).

El hecho de que exista un caso de un problema en el cual la solución es inmediata, no provoca que el coste de este algoritmo sea constante. Nunca podemos considerar el coste de un algoritmo en el caso más fácil puesto que este puede estar sujeto a suerte o ciertas condiciones que no se pueden dar siempre.

Existen una serie de reglas de complejidad a tener en cuenta, que facilitan la tarea de calcular la eficiencia:**[1]**

- Se ignoran las constantes multiplicativas o aditivas.
- La base del logaritmo no importa.
- *Suma de complejidades* es el máximo de entre todas ellas.
- *Composición de complejidades* es el producto de las complejidades.

2.4. ¿P = NP?

Los problemas según puedan ser resueltos, es decir, lograr una solución óptima, se pueden clasificar según la complejidad de los algoritmos que la logran.

Si la complejidad de los algoritmos es de las nombradas anteriormente, excepto la exponencial, se dice que los problemas que se resuelven son polinómicos, por lo que estos entran dentro de la categoría P.
Si no somos capaces de resolverlos por algoritmos de esa complejidad, es decir, no podemos encontrar la solución en un tiempo razonable, pertenecen a la categoría NP (Non deterministic Polynomial time).

La complejidad que se usa es la del algoritmo menos complejo que pueda resolver el problema. Evidentemente algoritmos más complejos pueden resolver problemas, pero se buscará el algoritmo menos complejo.

Intuitivamente podríamos dividir los problemas entre las dos categorías mencionadas anteriormente, diciendo que el conjunto de problemas P son aquellos que pueden resolverse en tiempo polinómico y el conjunto de problemas NP aquellos que no. Sin embargo, estas definiciones no son completas, pues no se ha conseguido demostrar que sean ciertas.
Siendo más rigurosos, NP es el conjunto de problemas de decisión que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista (dado el estado actual de la máquina y las variables de entrada, existe al menos un par de acciones posibles que esta puede tomar)[2].
Por otro lado, P engloba a aquellos problemas que pueden ser resueltos de manera eficiente en tiempo polinomial en una máquina determinista secuencial (dado el estado actual de la máquina y las variables de entrada, existe una única de acción posible que esta puede tomar, y cada acción es realizada una detrás de otra)[3], aunque existan problemas pertenecientes a P que no son tratables en términos prácticos. P es un subconjunto de NP, y se cree que es un subconjunto estricto, aunque aún no se haya demostrado.
Además, existen otras dos clasificaciones que pueden formar parte del conjunto NP:

- *NP-completo*, que contiene al conjunto de problemas, para los cuales no se ha conseguido encontrar algún algoritmo mejor que los que se basan en búsqueda exhaustiva.

- *NP-duro (Hard)*, que contiene al conjunto de problemas que son al menos tan difíciles como los pertenecientes al subconjunto NP. La definición precisa nos dice que un cierto problema X es NP-duro si existe un problema Y NP, de tal manera que Y puede ser transformado en X en tiempo polinómico. Esta afirmación estaría justificada porque si se puede encontrar algún algoritmo que sea capaz de resolver algún problema de NP-duro en tiempo polinómico, entonces será posible obtener un algoritmo que se ejecute en tiempo

polinómico para cualquier problema NP, transformando primero este a NP-duro.
Cabe apuntar que los problemas de NP-duro no tienen por qué estar en NP, es decir, no tienen por qué tener soluciones verificables en tiempo polinómico.

A continuación, presentamos un diagrama de Euler con las familias de problemas P, NP, NP-completo y NP-duro[2]:

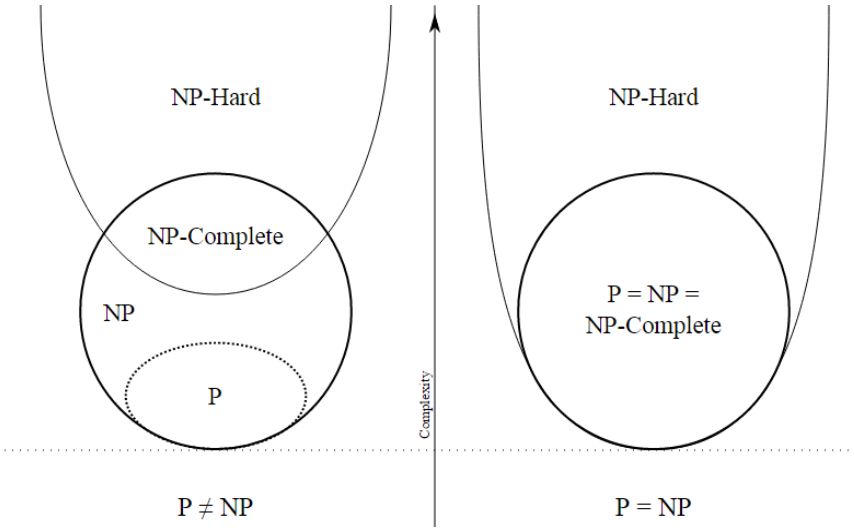


Figura 2.1. Diagrama de Euler de los problemas P, NP, NP-completo y NP-hard. Recuperada de https://en.wikipedia.org/wiki/NP_(complexity)#/media/File:P_np_np-complete_np-hard.svg

El problema está entonces en demostrar que un problema NP no se puede resolver como P, o en conseguir demostrar todo lo contrario, que un problema clasificado como NP se pueda convertir en P o equivalentemente, que todo problema pueda resolverse en tiempo polinómico.

Para intentar explicar esto de una manera más práctica, recurriremos a un ejemplo:

Resolver manualmente la raíz cuadrada de un número siempre debería llevar más tiempo que elevar ese número al cuadrado. Entonces llegamos a la conclusión de que resolver una raíz cuadrada (existe un método muy laborioso) es más costoso temporalmente que la operación inversa. Así pues, si un problema nos pide que comprobemos si un número determinado X es la raíz cuadrada de Z, lo que debemos hacer es idear un método para calcular la raíz cuadrada de Z, y por tanto el problema en cuestión tiene la complejidad de este método. Sin embargo, nos damos cuenta de que podemos resolverlo de otra forma mucho más simple, la cual es elevando al cuadrado a X y compararlo con Z, de forma que conseguimos reducir la complejidad del problema**[4]**. Algo parecido ocurre con los problemas NP, para los que se encuentran métodos nuevos para alcanzar la solución óptima de manera más eficiente, por lo que su complejidad se reduce y pasan a incluirse en la categoría P.

2.5. Tipos de problemas combinatorios

Según la modelización de la solución, podemos clasificar los problemas de la siguiente manera:

- *Permutación*: Dada una colección de datos, el objetivo es una reordenación de estos. Dentro de este tipo se encuentran problemas como el del TSP.

- *Binarios*: Se busca un subconjunto de elementos dado un conjunto más grande. Las variables de decisión toman valores binarios: "sí" o "no", "cierto" o "falso", ... Estos valores representan la pertenencia a la solución por parte de los datos. Dentro de este tipo podemos encontrar problemas como la Mochila 0-1 o el SAT3.

- *Enteros*: Se pide ponderar a los elementos del problema, por lo que los valores de las variables representan una cardinalidad en las variables de decisión, es decir, las variables de decisión toman valores enteros, Como por ejemplo en el problema de la *mochila entera*.

Además de esta clasificación simple, podemos encontrar la clasificación de los problemas de Karp:

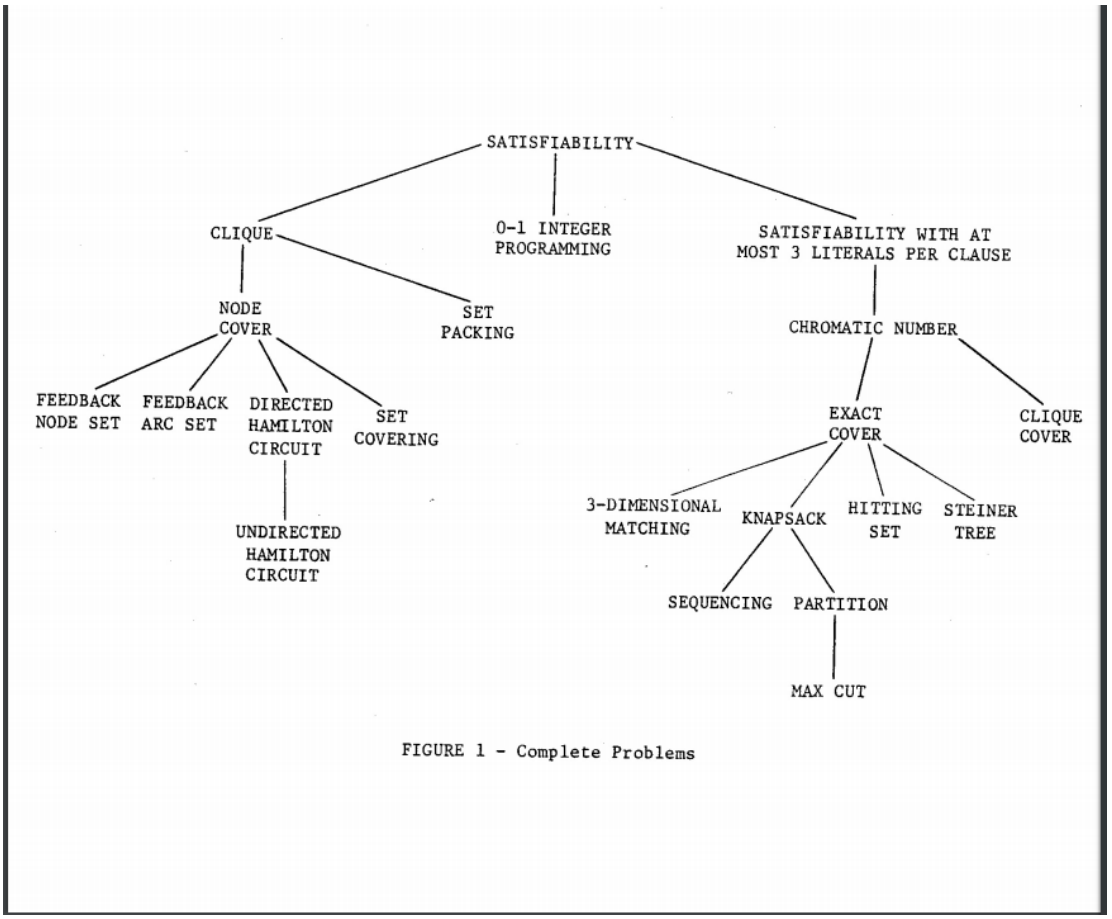


Figura 2.2. Clasificación de los problemas de Karp

2.6. Heurísticas y metaheurísticas

Las *heurísticas* son procesos simples que consiguen una buena solución inicial con una complejidad relativamente baja de manera sencilla y rápida en contrapartida a los métodos denominados exactos, que siempre han de devolver una solución óptima y factible. La mayoría de heurísticas están basadas en el sentido común, sin seguir un análisis formal de los problemas a resolver.

En el campo de la Inteligencia Artificial se usa el concepto de heurística aplicado al empleo del conocimiento en la realización de tareas**[5]**.

En lo que respecta a la Investigación Operativa, se habla de heurísticas para hacer referencia a procedimientos para la resolución de problemas de optimización.

Es normal también hablar de heurísticas cuando, sabiendo de antemano los entresijos de un problema, se realizan modificaciones en el proceso de resolución del mismo, que mejoran su eficiencia.

Las *metaheurísticas,* como su propio nombre parece indicar, se encuentran a un nivel más alto que las heurísticas, pues son una forma más inteligente de resolución de problemas, dando lugar a algoritmos que deben proporcionar soluciones muy cercanas a las óptimo, incluso pudiendo lograrlo.
Este término surgió por vez primera en el año 1986 en un artículo de Fred Glover sobre búsqueda tabú. Desde ese momento, han ido surgiendo multitud de propuestas para diseñar buenos métodos de resolución de problemas**[5]**.

Existen dos concepciones básicas sobre las metaheurísticas:

- Son procesos (iterativos) que guían y modifican las operaciones de las heurísticas para dar calidad a las soluciones.

- Toman como base procedimientos heurísticos inteligentes para ser aplicados de manera general a un gran número de problemas.

En muchos casos la estrategia consiste en tener una heurística para encontrar un punto bueno y después aplicar una metaheurística para mejorarlo más aún si cabe, para finalmente obtener una solución buena al problema al que nos enfrentamos.

2.7. Tipos de metaheurísticas

Para que un algoritmo sea clasificado como heurístico o metaheurístico, hemos de fijarnos principalmente en el modo en que estos buscan y construyen las soluciones. Se tienen cuatro familias principales de metaheurísticas[5]:

- *Metaheurísticas de búsqueda*: recorren el espacio de soluciones. Exploran las estructuras del problema valiéndose de transformaciones y movimientos.

- *Metaheurísticas de relajación*: son heurísticas que toman versiones relajadas, menos restrictivas y más fáciles de resolver que las del problema de optimización original para para obtener una solución.

- *Metaheurísticas constructivas*: para alcanzar una solución al problema, analizan y seleccionan minuciosamente las componentes de la misma.

- *Metaheurísticas evolutivas*: hacen evolucionar de forma simultánea a los valores de un conjunto de soluciones, de modo que en cada nueva iteración se van acercando cada vez más al óptimo del problema.

De entre las cuatro familias anteriores, las metaheurísticas de búsqueda representan uno de los grupos más a tener en cuenta, pues gracias a ellas se pueden utilizar estrategias para barrer el espacio de soluciones, transformando de manera "inteligente" soluciones de partida en otras, por lo que hacen posible encontrar muchos más puntos muy buenos del sistema en los que encontrar soluciones satisfactorias[5].

Existen 2 subgrupos en los que podemos dividir las metaheurísticas de búsqueda:

- *De búsqueda local*: está basada en el concepto de localidad o vecindad. Suelen ser algoritmos sencillos fáciles de entender. Consiste en modificar iterativamente y de manera inteligente una solución hasta que ninguna solución cercana mejore a la actual.

El hecho de ir descartando soluciones que no superan a la mejor obtenida hasta el momento, hace que estas metaheurísticas de búsqueda local se conviertan en algoritmos voraces.

El principal inconveniente de las búsquedas locales es que se pueden quedar ancladas en óptimos locales con facilidad.

- *De búsqueda global*: extienden su búsqueda al resto del espacio de soluciones, evitando de esta manera la localidad de las metaheurísticas de búsqueda local. Para conseguirlo, pueden valerse de las siguientes técnicas[5]:

- Se permite empeorar la solución actual para poder barrer todo el espacio de soluciones, lo cual evita el estancamiento en óptimos locales. Este empeoramiento se puede controlar y llevar a cabo principalmente de dos maneras:

    1. Memorizando recorridos del espacio de soluciones ya explorados, con el fin de evitar que la búsqueda se concentre en las mismas zonas.

    2. Regulando la probabilidad de aceptar transformaciones que no mejoren la solución actual. Para llevar a cabo esto, se hace uso de *criterios de aceptación estocásticos.*

- Reiniciando la búsqueda desde otra solución de arranque. Esta técnica es denominada *búsqueda por arranque múltiple* (MSS, Multi Start Search), las cuales siguen ciertas pautas para reiniciar de forma inteligente las búsquedas locales.

- Modificando la estructura del entorno mediante búsquedas por entorno variable (VNS, Variable Neighbourhood Search). Estas modifican en cada paso el tipo de movimiento, evitando que la búsqueda quede atrapada en una estructura de entornos rígida.

“Siste viator”.

Capítulo 3

Problemas

En este capítulo se presentarán los problemas estudiados y resueltos. Se repasará un poco de su historia, se definirá en qué consisten y además se proporcionará su correspondiente modelo matemático.

3.1. Traveling Salesman Problem

3.1.1. Historia

El origen del TSP está claramente marcado por la teoría de grafos, pues se basa en el concepto de ciclo hamiltoniano, el cual debe su nombre al matemático irlandés Sir William Rowan Hamilton. El problema fue definido en el siglo XIX por éste y por el matemático británico Thomas Kirkman[6].

Previamente, Euler y Vandermonde entraron en discusión sobre el problema del tour del caballo, que consistía en encontrar un ciclo hamiltoniano para un grafo cuyos vértices representan los 64 cuadrados del tablero de ajedrez, con dos vértices adyacentes si y sólo si un caballo puede moverse en un paso de un cuadrado a otro.

En el año 1832 la aplicación práctica con la que se suele explicar el planteamiento del problema fue mencionada en un libro para el éxito de los agentes viajeros impreso en Alemania. En este se comenta que, con una planificación adecuada del tour a realizar por el agente en cuestión, este podría ganar tiempo y maximizar el número de ciudades visitadas. En el libro también se incluyen ejemplos de viajes entre Alemania y Suiza, pero sin entrar en detalles matemáticos.

Al poco tiempo Hassler Whitney de la Universidad de Princeton introdujo el nombre de "Travelling Salesman Problem"[6], con el que el problema es conocido. En la década que va desde 1950 a 1960, la popularidad del problema fue aumentando entre el círculo de científicos de Europa y Estados Unidos y durante las décadas posteriores fue estudiado por investigadores procedentes de diversas ramas de la ciencia hasta hoy en día.

A continuación, se presentan algunos logros para diversas instancias del problema, que se han conseguido a lo largo del tiempo:

En 1954 Dantzig, Fulkerson y Johnson resolvieron un caso para 49 ciudades del TSP. Esto significa que no solo que obtuvieron la solución, si no que demostraron que era la mejor de un conjunto de 10 ^ 600 soluciones posibles.

En 2001 Applegate, Bixby, Chvátal y Cook resolvieron un caso para 15.112 ciudades de Alemania. Este caso fue resuelto en una red de máquinas en las universidades de Rice y Princeton. El tiempo total de cómputo fue de 22,6 años y la longitud total del tour fue de aproximadamente 66.000 km (un poco más de una vuelta y media a la tierra por el ecuador).

En 2004 Applegate, Bixby, Cook y Helsgaun resolvieron un caso para 24.978 ciudades de Suecia, con una longitud total del tour de aproximadamente 72.500 km.

En 2006, Cook y otros, obtuvieron un recorrido óptimo para 85.900 ciudades dado por un problema de diseño de microchip y actualmente es la instancia más larga resuelta.

3.1.2. Definición

Con el paso del tiempo, El TSP se ha convertido en un problema clásico de optimización combinatoria, clasificado dentro del tipo problemas NP-Completos, siendo uno de los más complejos de resolver de manera eficiente computacionalmente hablando.
Su forma más general consiste en: dadas n ciudades, recorrerlas todas realizando el tour más corto posible en función de la distancia que en cada caso separa una ciudad de otra. Se empieza por una ciudad cualquiera y se debe volver a la misma tras el recorrido.

Existen variantes de este mismo problema, como:

- *STSP* (*TSP simétrico*): Las ciudades a recorrer tienen la misma distancia entre cada par ciudades, de i a j y de j a i.

- *SCP (Steiner Cycle Problem)*: donde el tour realizado no tiene por qué contener todas las ciudades del conjunto. Existen penalizaciones o beneficios por visitar ciertas ciudades, lo que conlleva a visitar o descartar ciudades.

- *GTSP (Generalized Traveling Salesman Problem)*: las ciudades están agrupadas en distintas regiones. Se debe minimizar la distancia del tour, donde solamente se puede visitar una ciudad de cada región.

- *TSP con múltiples viajantes*: donde más de un viajante deben visitar ciertas ciudades, evitando las ciudades ya visitadas por otros.

entre las cuales, hemos elegido el *STSP*.

3.1.3. Modelización del problema

Si representamos el problema mediante un grafo no dirigido $G = (N, E)$, donde N es el conjunto de vértices y E el número de aristas. El coste de cada arista del grafo es c_e, donde $e \in E$.

Entonces si definimos al conjunto de aristas que parten desde el nodo i de la siguiente manera: $\delta(i) = \{[i,\ j] \in E : j \in N \wedge i \in N\}$, y tomamos como nodo de partida del tour el nodo 1, podemos sugerir el siguiente modelo matemático para el problema del STSP, con el cual hemos trabajado:

$$x_e = \begin{cases} 1 & \text{si la arista e forma parte de la solución óptima} \\ 0 & \text{en otro caso} \end{cases}$$

$$\text{mín} \sum_{e \in E} c_e x_e$$

Sujeto a las siguientes restricciones:

$$\sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in N$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall S \subset N$$

$$0 \leq x_e \leq 1$$

$$x_e \in \mathbb{Z}$$

3.2. Problema de la Mochila

3.2.1. Historia

Este es otro famoso problema que se encuentra dentro de la lista de los 21 problemas NP-completos realizada por Richard Karp en el año 1972. Ha sido intensamente estudiado desde mediados del siglo XX y se hace referencia a él en un artículo de 1987 del matemático inglés George Mathews Ballard.

El problema de la mochila se presenta como un subproblema de otros problemas más generales. Su modelización es sencilla, sin embargo, no es un problema tan fácil de resolver, aunque existen algoritmos que encuentran soluciones para tamaños grandes del problema**[7]**.

3.2.2. Definición

El problema de la mochila (*Knapsack Problem*), es un problema de optimización combinatoria y se define de la siguiente manera: se dispone de una mochila que cuenta con una cierta *capacidad*, y, por otro lado, se tiene un conjunto de *diferentes tipos de objetos*. Existe una cantidad de objetos de cada tipo, cada uno de los cuales con un *beneficio* y un *peso*.
Dados estos elementos del problema y sus propiedades, la dificultad del problema está en obtener el *subconjunto* de objetos, cuya suma de beneficios sea el máximo, sin sobrepasar nunca la capacidad de la mochila.

Al igual que en el *TSP*, existen también variantes para el problema de la mochila, entre las que podemos encontrar:

- *Problema de la Mochila 0-1*: Cuando existe 1 solo objeto por cada tipo.

- *Problema de la Mochila no acotado*: Cuando algún tipo contiene un número infinito de objetos.

- *Problema de la Mochila de múltiple elección*: Cuando estamos frente a un problema de Mochila 0-1, donde los objetos están subdivididos en clases, y solamente podemos seleccionar un objeto de cada clase.

- *Problema de la Mochila múltiple*: Cuando en un problema de Mochila 0-1 se tiene más de una mochila, cada una con su capacidad.

Nosotros realizaremos el estudio sobre un tipo de problema de Mochila0-1, concretamente el perteneciente la familia de los problemas de programación entera, donde los objetos del conjunto no se repiten y no se pueden partir.

3.2.3. Modelización del problema

Si únicamente existe un tipo de objeto al que denominamos como q y una capacidad C total en la mochila, podemos sugerir el siguiente modelo matemático para el problema de la Mochila 0-1:

$$x_i = \begin{cases} 1 & Si \;\; se \;\; selecciona \;\; el \;\; objeto \;\; i \\ 0 & Si \;\; no \end{cases}$$

$$\sum_{i \in q} b_i * x_i$$

Sujeto a:

$$\sum_{i \in q} c_i * x_i \leq C$$

$$\forall x_i \in \{0,\ 1\}$$

Donde:

$b_i$es el beneficio de cada objeto i-ésimo.
$c_i$es el coste de cada objeto i-ésimo.

"Una generación desaparece y otra ocupa su lugar, pero la tierra permanece eterna", película "La máquina del tiempo", 2002.

Capítulo 4

Métodos de resolución

En este capítulo se van a presentar los métodos exactos y las metaheurísticas estudiadas para la resolución de problemas. Se expondrán sus bases teóricas y se proporcionarán los esquemas básicos de la implementación de sus algoritmos.

4.1. Métodos exactos

Los métodos exactos por regla general realizan una búsqueda exhaustiva dentro del espacio de todas las soluciones parciales del problema a resolver, lo que conlleva a que su tiempo de ejecución para tamaños de entrada grandes, sea demasiado alto. Pese a esto puede ser una buena idea utilizarlos para tamaños de entrada reducidos y para realizar comparativas con otros métodos aproximados como los métodos metaheurísticos.

4.1.1. Fuerza bruta

Son algoritmos exactos, que consiguen la solución óptima siempre que esta exista probando cada una de las soluciones posibles del problema, por lo que también son conocidos como algoritmos de búsqueda exhaustiva o de búsqueda combinatoria.
Según el problema, se pueden adaptar para barrer una región determinada o todo el espacio de soluciones. En algunos problemas es útil limitar esta región únicamente a la *región factible*, en la cual se encuentran solo las soluciones factibles. Aunque también es posible barrer todo el espacio de soluciones, incluyendo las no factibles, detectando su no factibilidad y rechazándolas sin hacer valoraciones de la función objetivo.
Los algoritmos de fuerza bruta son fáciles de implementar, sin embargo, su gran inconveniente es que su tiempo de ejecución es proporcional al número de combinaciones posibles de soluciones candidatas del problema que estamos resolviendo.
Sin embargo, este tipo de algoritmos son usados cuando no hay otra opción como, por ejemplo, en el poco ético descifrado de contraseñas de wifi.

4.1.2. Vuelta atrás

Es un método parecido al anterior, solo que en este se va construyendo la solución poco a poco por fases. Entonces tendremos soluciones parciales, las cuales deberán de ser comprobadas en cada paso con el objeto de saber si podrían finalmente dar lugar a una solución satisfactoria. Durante este proceso, si se comprueba que una solución parcial es imposible que esté encaminada a convertirse en una solución final satisfactoria, se aborta la exploración del resto de soluciones pertenecientes al subárbol de búsqueda generado por dicha solución parcial. Cuando una solución candidata no es prometedora y es descartada, se realiza un retorno hacia la solución parcial viable que generó esta última. De esta forma se puede asegurar construir soluciones factibles.

Cabe apuntar que la manera en que los algoritmos de vuelta atrás realizan la búsqueda de soluciones están basadas en una dinámica de prueba y error y que hay que tener en cuenta que podemos enfrentarnos a ciertos problemas donde en cada etapa el número de pruebas a realizar crezca de manera exponencial, lo cual conlleva a una complejidad temporal alta, que se debe tratar de evitar siempre que se pueda.

Las soluciones factibles serán expresadas mediante una n-tupla $x_i, \ldots, x_n$, donde cada x_i viene a representar la decisión que se tomó en la fase i-ésima.

Existen dos tipos de restricciones cuando intentamos resolver un problema mediante el método de vuelta atrás:

1. *Restricciones explícitas*: que restringen los valores que pueden tomar cada uno de los componentes x_i de la n-tupla**[1]**.

2. *Restricciones implícitas*: representan las relaciones que se deben dar entre los posibles valores de cada componente x_i de la n-tupla solución para que estos puedan formar parte de ella y satisfacer la función objetivo**[1]**.

El esquema general de un algoritmo recursivo de vuelta atrás es el siguiente:

```
Algoritmo de vuelta atrás:
     Preparación de la búsqueda en el nivel k
     Mientras no se haya alcanzado el último hijo del
     nivel k, hacer:
          Solución[k] = siguienteHijoNivel(k)
          Si es solución válida(solución, k), entonces
               Si es solución completa(solución, k),
               entonces
                    Tratar solución
               Si no
                    VueltaAtras(solución, k + 1)
               Fin si
          Fin si
     Fin mientras
Fin
```

Podas

Una poda es un mecanismo que se vale de ciertos criterios para predecir cuándo una solución parcial en ningún caso va a llevarnos a obtener una solución completa satisfactoria, por lo que es necesario descartar la exploración del espacio de búsqueda a partir de dicha solución parcial.

Existen varios tipos de podas:

- *Podas de factibilidad:* son aquellas que evitan seguir construyendo soluciones para un problema concreto, es decir, una solución puede ser factible a un esquema de problema, pero para un caso concreto deja de ser factible, con lo que se puede parar.

- *Podas de optimización*: estas se hacen cuando la parte construida no puede alcanzar una solución mejor que una encontrada anteriormente.

- *Podas de estimación*: son similares a las podas de optimización, pero en este caso se basan en una estimación de *optimalidad* de las soluciones construidas a partir del fragmento de solución que se lleva construido hasta el momento. De esta forma, se puede establecer una medida estimada del caso mejor, y compararla con la medida de la solución óptima lograda hasta ahora. De manera que, si no se puede mejorar la solución que se está construyendo, esta se descarta.

Existen otros tipos de podas como las podas *alfa-beta*, utilizadas en *árboles de juego*.

4.1.3. Ramificación y poda

Es un método similar al de vuelta atrás, solo que, en vez de tener un árbol de búsqueda en sentido abstracto, establecemos esta estructura mediante nodos en memoria. A su vez se tiene una cola de prioridad en la que se almacenan los nodos sin usar y se guardan otros nodos ya usados. Dentro de estos nodos se almacenan las soluciones y las estimaciones a usar.
Estos nodos se pueden clasificar según el estado de uso:

- *Muerto*: nodo que no puede mejorar la solución actual y desde el que no se ramifican nodos hijos.

- *Pendiente*: está en la cola de prioridad esperando a ser valorado.

- *Explorado*: nodo cuyos hijos han sido valorados ya y no queda ninguno por explorar.

La principal diferencia entre los algoritmos de ramificación y poda y los de vuelta atrás es la forma en que recorren el espacio de soluciones. Mientras que en vuelta atrás se realiza un recorrido en profundidad del árbol generado, en ramificación y poda en cada paso se toma el nodo más prometedor con el que se pueda alcanzar una solución factible. La gestión de estos nodos se lleva a cabo mediante una cola de prioridad[8].

El esquema general de un algoritmo de ramificación y poda es el siguiente:

```
Algoritmo de ramificación y poda:
     Inicialización: crear la cola de prioridad y añadir
     en ella el primer nodo raíz.

     Mientras la cola de prioridad no esté vacía, hacer
          Sacar el primer (mejor) nodo,
          Si es solución válida y estimación < coste
          mejor, entonces
               Si esta solución es completa, entonces
                    Añadir solución a la cola de
                    prioridad
               Si no
                    Continuar la búsqueda en el siguiente
                    nivel
               Fin si
          Fin si
     Fin mientras
Fin
```

4.1.4. Programación dinámica

Técnicas algorítmicas como las presentadas hasta ahora solo pueden ser utilizadas en algunos tipos de problemas. Ante esto los algoritmos de programación dinámica se presentan como una alternativa de alta aplicabilidad.

La programación dinámica es un potente paradigma algorítmico en el que un problema es resuelto mediante la identificación de un conjunto de subproblemas, abordando uno a uno desde el más pequeño. Se utilizan las soluciones de subproblemas más pequeños para resolver subproblemas más grandes, para obtener finalmente una solución completa final del problema original, por lo que se puede decir que se sigue un esquema recursivo de resolución. Es aquí donde radica principalmente la poca adaptabilidad a ciertos problemas por parte de algunos métodos algorítmicos, como por ejemplo *Divide y vencerás*, donde en muchas ocasiones se realizan demasiadas llamadas recursivas, con el consecuente aumento en el coste de la eficiencia de sus algoritmos.

La programación dinámica utiliza una tabla donde se almacenan una única vez los resultados de cada subproblema más pequeño ya resuelto, evitando de esta manera repetir cálculos innecesarios.

Aunque la programación dinámica se aplique a la resolución de un gran número de problemas, no siempre será posible aplicar este método. Para poder aplicarla al problema que deseamos resolver, este ha de ajustarse al *principio de optimalidad de Bellman,* que dice que *"Dada una secuencia óptima de decisiones, toda subsecuencia de ella, es a su vez óptima"*.

En este caso no presentaremos un esquema general de la implementación de los algoritmos de programación dinámica, pero si se darán unas pautas a seguir para su diseño**[8]**:

- Especificación recursiva de la función que expresa el problema que vamos a resolver.

- Determinar las ecuaciones de recurrencia para calcular la función representativa del problema.

- Comprobar el alto coste del cálculo de la función debido a la repetición de subproblemas a resolver.

- Representación de la función del problema mediante una tabla.

- Inicializar dicha tabla con los casos base de la función.

- Sustituir las llamadas recursivas de las ecuaciones de recurrencia por consultas a la tabla.

- Planificar el orden en que la tabla será llenada, para obtener la solución del problema.

4.2. Métodos voraces

Los algoritmos pertenecientes a esta metodología suelen ser fáciles de diseñar e implementar, sin embargo, no siempre aseguran obtener el óptimo, aunque suelen ofrecer soluciones bastante razonables. Son aplicados generalmente a problemas de optimización.

Sus características fundamentales son**[8]**:

- Al principio se tiene un *conjunto de candidatos* y a medida que se va realizando la ejecución del algoritmo, se generan dos subconjuntos más: uno para los *candidatos seleccionados* y otro para los que han sido *definitivamente rechazados*.

- Los candidatos más prometedores se van seleccionando según una *función de selección*.

- Se comprueba si un candidato es factible para formar parte de la solución que se está construyendo mediante un test de factibilidad.

- Por último, existe un *test de solución* para verificar si una solución parcial es ya una solución completa.

En cuanto a su funcionamiento, los algoritmos voraces añaden al conjunto de seleccionados aquellos candidatos devueltos por la función de selección, siempre y cuando pasen con éxito el test de *satisfactibilidad*. Serán rechazados en caso contrario.

La filosofía que se sigue es la de tratar una única vez a cada candidato, sin replantearse una decisión ya tomada.

El esquema general de un algoritmo voraz es el siguiente:

```
Algoritmo voraz:
    Inicialización: conjunto candidatos, solución
    Mientras existan candidatos y no se haya obtenido
    una solución, hacer
        Seleccionar candidato y eliminarlo del conjunto
        de candidatos
        Si el candidato es factible, entonces
            El candidato pasa a formar parte de la
            solución
        Fin si
    Fin mientras
Fin
```

4.3. Métodos aleatorios

Los algoritmos pertenecientes a este método suelen ser simples y utilizan un cierto grado de aleatoriedad como parte de su lógica. Dicha aleatoriedad puede presentarse en la entrada de los datos o en el comportamiento al azar de la ejecución de estos algoritmos.

Dado un número máximo de iteraciones, se generan tantas soluciones como se indique, y de ellas, se va almacenando la mejor.

4.4. Metaheurísticas

Al ser los algoritmos heurísticos resultado de la inspiración de sus creadores y al intervenir en su diseño diversos elementos, no resulta fácil realizar una clasificación de tipos. Presentaremos en esta sección una serie de metaheurísticas, las cuales hemos estudiado durante el desarrollo de este trabajo.

4.4.1. Metaheurísticas evolutivas

Están basadas en un aspecto de la naturaleza, en este caso, en la evolución de las especies. Dentro de esta evolución podemos destacar varias teorías: Mendel, con la cual se inició el estudio de la herencia genética por medio de la cual ciertos individuos (guisantes en su origen), transmitían ciertos rasgos genéticos a sus descendientes; Darwin y la selección natural, por la cual, los individuos más capacitados son los que sobreviven en la naturaleza y gracias a esta supervivencia, son los que determinan los individuos de la generación siguiente. Además, hay que añadir a Lamarck, que, aunque su teoría se basaba en que la necesidad crea el órgano, podemos coger una pequeña adaptación de dicha teoría y aplicarla a la programación[9].

Dentro de este contexto, se toma conocimiento de diversos campos y teorías y se reúnen en la programación evolutiva y los algoritmos genéticos.

4.4.1.1. Algoritmos genéticos

El principal elemento de los algoritmos genéticos es una población de individuos (o soluciones) sobre la que se realiza una selección de los mejores, con los cuales se crea una descendencia que sigue ciertas reglas de genética con las que los hijos comparten características de los padres.

Si a su vez, a la hora de crear estos nuevos individuos, además, los mejoramos con una búsqueda local, obtenemos los denominados *algoritmos meméticos*.

Un algoritmo genético simula una solución de un problema como un individuo constituido por un genoma o conjunto de genes. Las soluciones que se manejan en el algoritmo se encapsulan en un individuo, el cual se caracteriza por componerse por genes y por tener una serie de operadores.

A partir de estos individuos, se van creando otros nuevos, por medio de la recombinación genética, modificando algún gen, seleccionando los mejores como en la selección natural propuesta por Darwin. De manera que según se van creando individuos, se van conservando los más aptos y son estos los elegidos para tener descendencia[9] [10].
Para ello se usan poblaciones de individuos, que son conjuntos de esas soluciones sobre las cuales se opera.

Además, para valorar la *aptitud* de un individuo se usan diversos mecanismos, los cuales suelen ir ligados al tipo del individuo o al problema. Dicha aptitud se denomina *Fitness* y suele usar la función *objetivo* del problema.

El esquema general de un algoritmo genético es el siguiente[9] [10]:

1. Se crea una población inicial.
2. De esta población, se eligen una serie de individuos "padres" para combinarse, es decir, para crear una descendencia o tener unos "hijos". Estos individuos padres pueden ser elegidos por diversos criterios: de manera aleatoria, elegir los mejores, ….
3. Elegidos todos o algunos (2) padres, cruzarlos para crear hijos, para lo que se usan diversos mecanismos de combinación o cruce.
4. Sobre los hijos, se aplica una mutación dada una determinada probabilidad, con el fin de conservar una diversidad genética. La mutación consiste en variar algún gen en los hijos.
5. En caso de ser *memético*, mejorar los hijos por medio de búsquedas locales.
6. Introducción de los hijos en la población, por diferentes mecanismos de sustitución de los padres.
7. Repetir los pasos 2 a 4, hasta el criterio de finalización, el cual puede ser determinado por el número de generaciones, o por un criterio de parada basado en óptimos locales.

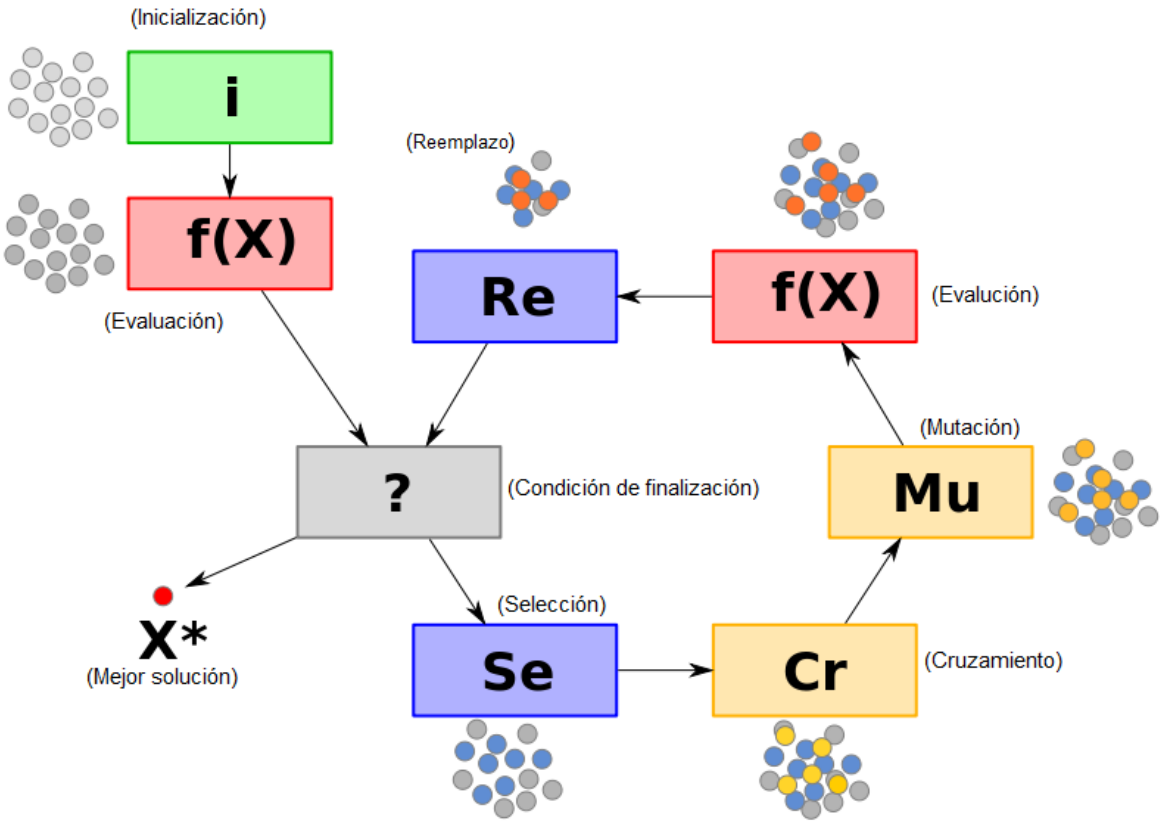


Figura 4.1. Algoritmo genético. Recuperada de https://es.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico#/media/File:Evolutionary_algorithm.svg

Por otro lado, los parámetros típicos de los algoritmos genéticos son:

- *Tasa de Mutación*: probabilidad de que un individuo mute, o porcentaje de mutación de un individuo en sus genes. Debe ser muy pequeña para no modificar demasiado los individuos, pero debe existir, para conseguir diversidad genética y no estancarse en máximos locales.

- *Tasa de Cruce*: probabilidad de cruzamiento entre los padres. Si es 0, los hijos son los padres, si es 1, se cruza. Debe ser alta para crear nuevos individuos.

- *Elitismo*: existe elitismo si conserva el mejor individuo de la anterior población.

- *Número de generaciones*: número de generaciones a crear.

- *Tamaño de la población*: número de individuos en cada población.

Hay dos tipos de algoritmos genéticos: *Generacionales* y *estacionarios*. Además, existe un tercer tipo compuesto, el *modelo de islas*[11].

4.4.1.2. Algoritmos genéticos generacionales

En este tipo de algoritmos genéticos, en cada generación de individuos, crearemos una nueva población con los descendientes de la generación de los padres. La nueva población de hijos sustituirá por completo a la de los padres.

El elitismo en un algoritmo genético generacional implica almacenar en la nueva población el mejor individuo de la anterior. Según ciertas personas el elitismo sólo se considera una característica a incorporar en el algoritmo, otros lo consideran un subtipo de algoritmo genético generacional.

4.4.1.3. Algoritmos genéticos estacionarios

Los *estacionarios*, al contrario que los generacionales, no crean una nueva población para que sustituya a la antigua, sino que la población se conserva y se modifica. Los individuos nuevos se pueden introducir en la población sustituyendo a individuos antiguos que ya pertenecían a ella[11].
Se usan algoritmos de selección para sustituir los individuos y así mantener una población buena, pero diversa. Según van pasando las generaciones, se crean nuevos individuos, (tanto peores como mejores de los que existen en la población actual) y se usan criterios de reemplazo para sustituir a los que ya pertenecían a la población por los nuevos (mejores). De esta manera, la población contendrá sólo los más aptos. Aunque, para preservar cierta diversidad genética, podrán ocurrir excepciones.

4.4.1.4. Codificación de la solución (individuo)

Como hemos dicho, un individuo encapsula una solución. Esta solución viene almacenada en una estructura denominada genoma. El genoma es una tupla de tipo T, siendo T un número entero real, binario (0 o 1) o booleano (true/false), o cualquier otro tipo. Estas tuplas suelen ser de longitud fija en cada problema y representan la dimensión del problema, aunque a veces es posible que en un mismo algoritmo se den individuos de diferente tamaño.

Algo importante de un problema es elegir este tipo T y el tipo de individuo, esto se llama codificación del individuo. Es útil saber la diversidad de elecciones que tenemos para representar a un individuo. Esta representación estará dotada de características y restricciones.

Además, diremos que, si un individuo cumple las restricciones, es un *individuo factible*, que implicará que la solución que encapsula sea factible.

Según Fang (1994) las normas básicas a tener en cuenta para el diseño de una codificación óptima se pueden resumir en**[10]**:

1. Cada solución del problema tendrá su correspondiente cadena codificada.

2. Para cada cadena codificada producida mediante los operadores genéticos existirá su correspondiente solución decodificada.

3. Codificación y solución se corresponden una a una, es decir no habrá redundancia.

4. La codificación deberá permitir heredar de padres a hijos las características inherentes a los primeros.

4.4.1.5. Función de fitness

La función de fitness o aptitud define la bondad de un individuo, es decir, lo buena que es la solución que encapsula dicho individuo. Para calcular este valor adaptaremos la función objetivo al tipo de individuo que tenemos. Para recordar la función objetivo podrá ser de *maximización* o de *minimización* según el tipo de problema.

Sin embargo, si tenemos un individuo *no factible*, habrá que usar algún criterio para favorecer a los que son factibles que a los que no lo son**[12] [13]**. Para ello utilizaremos penalizaciones, entre las que podemos contar con una penalización constante, una penalización anuladora, penalizaciones relativas, por ejemplo, según el grado de *infactibilidad*, o según la generación, ...

4.4.1.6. Poblaciones

Para almacenar los individuos, usaremos unas estructuras denominadas poblaciones. Estas utilizarán un mecanismo de ordenamiento para favorecer ciertos cálculos. Se dispondrán de mecanismos de reemplazo para sustituir los individuos nuevos por otros antiguos según se requiera. Además, también se podrá controlar la repetición de individuos.

Según el criterio de ordenación, podemos encontrar poblaciones *desordenadas,* que no necesitan ordenar sus individuos, poblaciones *ordenables,* que en algún momento permiten ordenar sus individuos, poblaciones *ordenadas*, que según insertamos un individuo, lo coloca en su posición correspondiente. Y según el mecanismo de reemplazo, podrán ser generacionales, que no reemplazan o estacionarias, que, si lo hacen, y dentro de estas podremos encontrar sustitutivas o reductoras, las cuales incluyen a su vez distintos tipos y criterios de reemplazo.

Existen muchas maneras de crear una población inicial, entre ellas aleatoria, uniforme, aleatoria parcial e invertida**[10]**. Pero según Michalewicz (1995) la forma más eficaz para crearla es la generación aleatoria.

4.4.1.7. Operadores de selección

Establecen mecanismos para elegir los padres con los que crear descendencia. Tenemos los siguientes**[10] [11]**:

- *Ruleta*: Dada una población (no tiene por qué estar ordenada), se simula una ruleta que se divide en tantos sectores como individuos, cuyos tamaños serán proporcionales al fitness que tiene cada individuo.

$$p_j = \frac{f_j}{\sum_i^n f_i}$$

- *Torneo*, del cual existen dos modalidades:

  1. *Determinístico:* se eligen p individuos, p normalmente 2, y se elige al mejor de ellos, es decir, el que mejor fitness tenga.

  2. *Probabilístico:* se eligen dos individuos, y se da una probabilidad p, la cual simboliza la probabilidad de elegir el mejor. Dicha probabilidad p se encuentra entre 0.5 y 1, de manera que se da primacía al mejor, pero podría darse el caso de elegir el peor de los dos. Esto aumenta la diversidad.

- *Ranking*: similar al operador de ruleta, pero requiere que los individuos estén ordenados según su fitness.

4.4.1.8. Operadores de cruce

Una vez elegidos los padres, procedemos a crear los hijos. Para ello, establecemos criterios para que intentar aprovechar lo mejor de cada padre y transferirlo a los hijos, por lo que usaremos patrones que encontraremos en los genomas de los padres.
Generalmente se usan dos padres, sin embargo, existe el cruce *multiparental*, el cual usa más de dos padres. Al igual que podemos tener diferente número de padres, podemos también tener distinto número de hijos. Podemos crear uno por padre, o dos por padre.

Existen diversas formas de realizar un cruce, entre los que destacamos:

- *Cruces de corte*, en los que establecemos puntos de corte y escogemos que parte utilizar de cada padre en cada hijo. Dentro los cruces de corte, los operadores sobre los que más hemos investigado son:

  - *SPX (cruzador de punto de corte simple)*: coge dos padres de longitud n y elige un punto de corte entre 0 y n. Este punto separará los padres en *cabeza* y *cola*. Para combinarlos, los hijos se compondrán de la cabeza del uno y la cola del otro.

  - *DPX (cruzador de punto de corte doble)*: coge dos padres de longitud n y elige dos puntos de corte entre 0 y n. Estos puntos separarán los padres en *cabeza, cuerpo* y *cola*. Para combinarlos, los hijos se compondrán de la cabeza y la cola de uno y el cuerpo del otro.

  - Cruzador en n Puntos: similar a los cruzadores *anteriores*. Se establecen n puntos de corte y los padres se dividirán en partes de esos puntos. Para combinarlos, los hijos se compondrán de estas partes, eligiendo cada vez una parte de un padre y la siguiente del otro.

  - *UPX (cruzador de punto uniforme)*: dados dos padres, se recorrerá el genoma de un hijo y se elegirá de forma arbitraria si el gen de cada posición i vendrá del padre 1 o del padre 2. Para esto se puede usar una máscara predeterminada o crear una. Esta máscara se compondrá de valores booleanos, los cuales se indicarán según el siguiente criterio: Si la máscara en la posición i es true, el hijo 1 cogerá el gen i del padre 1, y en caso contrario lo cogerá del padre 2. De forma análoga se creará el hijo 2, pero con el criterio invertido.

- *Cruces con permutaciones*: son utilizados sobre individuos (soluciones) representadas mediante permutaciones. *Los operadores de cruce basados en permutaciones* sobre los que más hemos investigado son:

  - *PMX (Partially Mapped Crossover)*: Una parte de la ristra que representa a uno de los padres, se hace corresponder con una parte de igual tamaño de la ristra del otro padre, intercambiándose la información restante.

Por ejemplo, si consideramos los dos padres siguientes:
(1 2 3 4 5 6 7 8) y (3 7 5 1 6 8 2 4), el operador PMX crea descendientes de la siguiente manera:

En primer lugar, selecciona con probabilidad uniforme dos puntos de corte a lo largo de las ristras que representan a los padres. Supongamos que el primer punto de corte se selecciona entre el tercer y el cuarto elemento de la ristra, y el segundo entre el sexto y el séptimo elemento:
(1 2 3 | 4 5 6 | 7 8) y (3 7 5 | 1 6 8 | 2 4).
Se considera que existe una correspondencia biunívoca entre los elementos que forman parte de las subristras comprendidas entre los puntos de corte. En nuestro ejemplo, las correspondencias establecidas son las siguientes: 4 ↔ 1, 5 ↔ 6 y 6 ↔ 8. A continuación la subristra del primer padre se copia en el segundo hijo. De forma análoga, la subristra del segundo padre se copia en el primer hijo, obteniéndose: descendiente 1: (x x x | 1 6 8 | x x) y descendiente 2: (x x x | 4 5 6 | x x). En el siguiente paso el descendiente
i-ésimo (i=1,2) se rellena copiando los elementos del i-ésimo padre.

En el caso de que un gen esté ya presente en el descendiente, se reemplaza teniendo en cuenta la correspondencia anterior. Por ejemplo, el primer elemento del descendiente 1 será un 1 al igual que el primer elemento del primer padre. Sin embargo, al existir un 1 en el descendiente 1 y teniendo en cuenta la correspondencia 1 ↔ 4, se escoge la ciudad 4 como primer elemento del descendiente 1.

El segundo, tercer séptimo elementos del descendiente 1 pueden escogerse del primer padre. Sin embargo, el último elemento del descendiente 1 debería ser un 8, ciudad ya presente. Teniendo en cuenta las correspondencias 8 ↔ 6, y 6 ↔ 5, se escoge en su lugar un 5.
De ahí que descendiente 1: 4 2 3 | 1 6 8 | 7 5). De forma análoga, se obtiene: descendiente 2: (3 7 8 | 4 5 6 | 2 1).

- *CX o de ciclos* (Oliver y col., 1987): el operador CX crea un descendiente a partir de los dos padres, de tal manera que cada posición se ocupa por el correspondiente elemento de uno de estos padres. Por ejemplo, considerando los padres (1 2 3 4 5 6 7 8) y (2 4 6 8 7 5 3 1), escogemos el primer elemento del descendiente bien del primer elemento del primer padre o del primer elemento del segundo padre. Por tanto, el primer elemento del descendiente debe ser un 1 o un 2.

Para empezar otro ciclo, se hace igual: se busca el primer hueco del descendiente, y se consulta los genes de esa posición en ambos padres y se elige uno de los dos.

4.4.1.9. Operadores de mutación

Modifican un individuo con el objetivo de aumentar la diversidad. Varían el contenido del genoma, variando uno o varios genes. Cambiando su valor, produciendo una reordenación, ...

El operador más común es el operador de individuos booleanos basado en el inverso (*Flip*), el cual consiste en invertir uno o varios valores del genoma. Existen otros basados en una reordenación de un segmento del genoma. Dicha reordenación puede ser aleatoria (DM), puede hacerse mediante una inversión del orden (*IVM*), insertando un gen en medio y desplazando el resto (*ISM*), se puede intercambiar la posición de dos genes (*OBM*), **...[11]**

4.4.1.10. Operadores de vecindad

Similares a los de mutación, pero el objetivo en este caso es mejorar la progenie. El mecanismo es el mismo, solo que lo van modificando hasta que se mejora el individuo, o lo modifican de todas las formas posibles dentro de una región de vecindad, conservando la mejor modificación. Para ello se hace una búsqueda local con los métodos de optimización ya explicados en cada problema, como, por ejemplo, 2-Opt.

4.4.2. Metaheurísticas inspiradas en colonias de hormigas

Se basan en el comportamiento que adoptan las hormigas a la hora de encaminarse hasta su fuente de alimento y volver al hormiguero. Este comportamiento se puede resumir en lo siguiente: cuando las hormigas salen hacia donde está su alimento, se mueven en un inicio siguiendo caminos aleatorios y depositando sus feromonas sobre ellos. Tras esto, otras hormigas que tengan que hacer lo mismo, optarán por seguir caminos más cortos, es decir, caminos con mayor concentración de feromonas, pues estos son los candidatos más atractivos a ser recorridos por las hormigas. De este modo llegará un momento en el que todas las hormigas consigan ir por el camino óptimo.
Además, las feromonas pueden evaporarse de manera que un camino podría perderse al ya no ser visitado por más hormigas.

La siguiente figura ilustra esta dinámica: en un principio una hormiga vuelve al hormiguero después haber ido a recoger alimento para llevarlo hacia éste por un camino seleccionado aleatoriamente sobre el que va depositando sus feromonas. Después más hormigas salen también a recoger más alimento, siguiendo algunas el mismo camino que la primera hormiga y depositando sus correspondientes feromonas. Por el contrario, otras deciden seguir caminos más cortos. Es lógico pensar que mientras más cortos sean los caminos, más concentración de feromonas tendrán estos a largo plazo, ya que las hormigas que decidan ir por ellos podrán ir y volver al hormiguero más rápidamente, aumentando la concentración de feromonas.
Finalmente, todas las hormigas atraídas por la mayor concentración de feromonas del camino más corto se decidirán únicamente por este para llegar hasta su destino. Como consecuencia la concentración de feromonas de los demás caminos poco a poco irá desapareciendo hasta evaporarse

totalmente, al ya no ser recorridos por las hormigas.

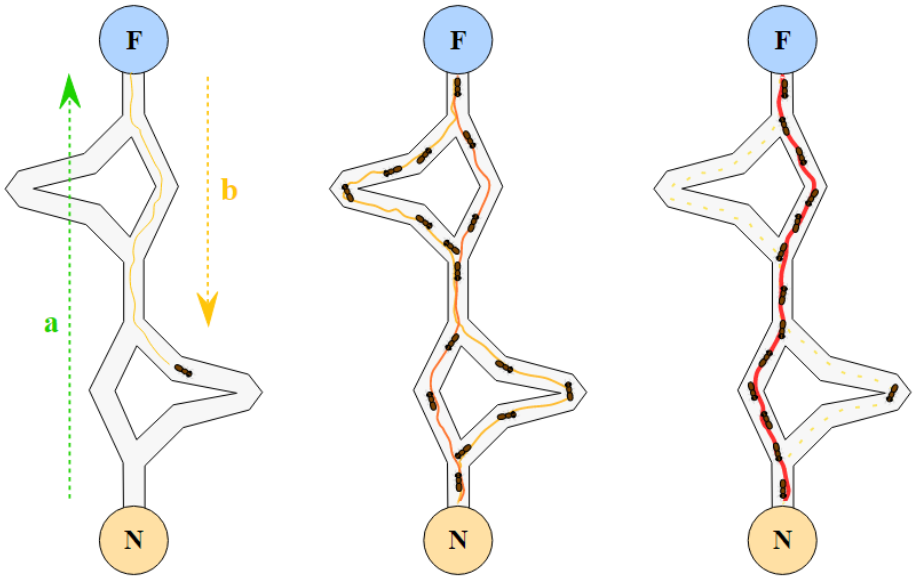


Figura 4.2. Comportamiento de las hormigas en búsqueda de su fuente de alimento. Recuperada de https://es.wikipedia.org/wiki/Algoritmo_de_la_colonia_de_hormigas#/media/File:Aco_branches.svg

4.4.2.1. S-ACO (Simple-Ant Colony Optimization)

El concepto de colonias de hormigas llevado al mundo de la algoritmia es usado tanto para encontrar caminos de coste mínimo como para resolver otro tipo de problemas donde se usan grafos para su representación.

Se ejecutan una serie de *hormigas virtuales* que buscan un camino de manera aleatoria. La tarea principal de cada una de ellas es la de encontrar el camino más corto que separa un par de nodos en un grafo que representa el problema a resolver. El algoritmo más básico ACO (S-ACO) consistirá exactamente en esto, tomándose uno de los nodos como origen y otro como destino, siendo la longitud del camino más corto el que viene dado por el número total de saltos realizados.

Cada arista (i, j) del grafo tiene asociada una variable τ_{ij} que viene a representar un rastro de feromonas artificiales. Estos rastros de feromonas artificiales guían y son esparcidos por las hormigas virtuales. Las hormigas estiman la utilidad de cada arista para construir buenas soluciones en función de la concentración de feromonas que estas presentan.

Cada hormiga aplica paso a paso decisiones según probabilidades para construir soluciones del problema. En cada nodo o arista saliente se almacena información para que las hormigas decidan de manera estocástica cual será el siguiente nodo al que saltarán. La regla de decisión de una hormiga k situada en un nodo i usa el rastro de feromonas artificial τ_{ij} para calcular la probabilidad con la cual se debería elegir un nodo $j \in N_i$ como el siguiente nodo al que saltar, donde N_i es el conjunto de nodos vecinos del nodo i a distancia 1.

$$p_{ij}^{k} = \begin{cases} \tau_{ij} & si \;\; j \in N_i \\ 0 & si \;\; j \notin N_i \end{cases}$$

Mientras construyen una solución, las hormigas depositan información de feromonas en las aristas que van utilizando. En los algoritmos S-ACO las hormigas depositan una cantidad constante $\Delta\tau$ de feromonas. Consideremos una hormiga que en un cierto instante t se mueve desde el nodo i hasta el nodo j, lo que provocará que el valor de τ_{ij} de la arista (i, j) cambie de la siguiente manera:

$$\tau_{ij}(t) \leftarrow \tau_{ij}(t) + \Delta_{\tau}$$

Esto aumentará la probabilidad de que otras hormigas vuelvan a utilizar en el futuro la misma arista.

Con el fin de evitar una pronta convergencia de las hormigas hacia un camino subóptimo, es decir, el estancamiento en óptimos locales, se añade un mecanismo de evaporación de feromonas virtuales, que servirá como mecanismo de exploración. De esta manera la intensidad de dichas feromonas va decreciendo automáticamente, favoreciendo la exploración de otras aristas durante todo el proceso de búsqueda del camino óptimo. La evaporación es llevada a cabo de una manera simple, con el decremento exponencial de los rastros de feromonas

$$\tau \leftarrow (1 - \rho)\tau, \rho \in (0, 1]$$

en cada iteración del algoritmo.

El esquema general de un algoritmo S-ACO es el siguiente:

```
Algoritmo S-ACO:
     Inicializar rastros de feromonas

     Mientras no se llegue a criterio de parada,
hacer
          Realizar recorridos
          Analizar rutas
          Actualizar rastros de feromonas
     Fin mientras
Fin
```

Para concluir, diremos que los algoritmos S-ACO al ser los más básicos, tienen una serie de limitaciones, como que solo sirven para resolver problemas de caminos mínimos sobre grafos.

Pero son la base sobre la que se construyen otros algoritmos también inspirados en colonias de hormigas, añadiendo estas otras mejoras.

4.4.2.2. Metaheurística ACO

Si consideramos la modelización de un problema de optimización discreta mediante un grafo, las soluciones obtenidas mediante ACO pueden ser representadas en términos de caminos factibles en dicho grafo[14].

Los algoritmos de la metaheurística ACO pueden ser utilizados para encontrar secuencias de caminos de coste mínimo que sean factibles respecto a ciertas restricciones. Se dispone de una colonia de hormigas que colectivamente resuelven el problema de optimización considerado mediante el grafo que lo modela. La información recogida por las hormigas durante el proceso de búsqueda es codificada en rastros de feromonas τ_{ij} asociada a cada arista (i, j).

Las hormigas poseen una memoria donde se almacena el camino por el que han ido hasta llegar al nodo de destino, y que les servirá como recordatorio para poder volver hacia atrás. Cuando estas vuelven hacia atrás van dejando rastros de feromonas en las aristas por las que van pasando. La memoria mencionada viene a ser una lista de los nodos visitados y que representa la solución construida por la hormiga en cuestión.

Al inicio, se asocia una cantidad constante de feromonas a un subconjunto de aristas del grafo. Dependiendo de la representación del problema elegido, estos rastros de feromonas pueden ser asociados a todas las aristas del grafo o únicamente a un subconjunto de ellas. Las aristas pueden también tener asociadas un cierto valor de la heurística η_{ij} que puede representar información a priori sobre la definición de la instancia del problema, o información en tiempo de ejecución proveniente de otra fuente distinta a la de las hormigas[14].

Al contrario que en los algoritmos S-ACO, donde las hormigas depositan una cantidad constante de feromonas, ahora dicha cantidad depositada en cada camino que atraviesan es proporcional al coste de estos, de manera que depositarán mayor cantidad de feromonas en las soluciones de coste más bajo.

Cuando una arista de un camino es atravesada por una hormiga k, la cantidad de feromonas de esta se ve incrementada en la cantidad de feromonas que acaba de ser depositada por la hormiga.

$$\tau_{ij} \leftarrow \tau_{ij} + cantidadFeromonas_{ij}^{k}$$

De manera que se aumenta el atractivo de esa arista para ser utilizada en el futuro por otras hormigas.

Además, también se incorpora el mecanismo de evaporación de feromonas, con lo que se reduce la probabilidad de elegir soluciones de baja calidad.

La siguiente expresión nos dará probabilidad de que una hormiga k situada en el nodo i salte al nodo j:

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}}{\sum_{l \in N_i^k} \tau_{il}} & si \quad j \in N_i^k \\ 0 & si \quad j \notin N_i^k \end{cases}$$

Donde:

N_i^k representa al conjunto de nodos vecinos a distancia 1 del nodo i sobre el que se encuentra situada la hormiga k.

Como ya se dijo, existen varios tipos de algoritmos basados en sistemas de colonias de hormigas, entre ellos el elitista, en el que solo la hormiga con el mejor recorrido, es decir, el más corto, es la que deposita las feromonas.

4.4.3. Simulated annealing

Simulated annealing (Enfriamiento simulado) es una metaheurística de búsqueda local que puede ser aplicada a problemas de optimización combinatoria. Está inspirada en el proceso físico de *templado de los metales,* en el que son calentados hasta que alcanzan altas temperaturas, para ser después enfriados de manera progresiva y controlada. Con esto se consiguen estados de baja energía, de manera que las partículas de dichos metales se recolocan en nuevas configuraciones.

Los algoritmos seleccionan candidatos de forma aleatoria y se permiten soluciones degradadas, pero siempre con una probabilidad de aceptación que dependerá de dos factores:

- La temperatura T
- La diferencia de valores entre funciones objetivo δ

Es decir, de la denominada *distribución de Boltzman:*

$$exp(\frac{-\delta}{T})$$

En la siguiente gráfica se puede apreciar la evolución de la probabilidad de aceptación en función de δ:

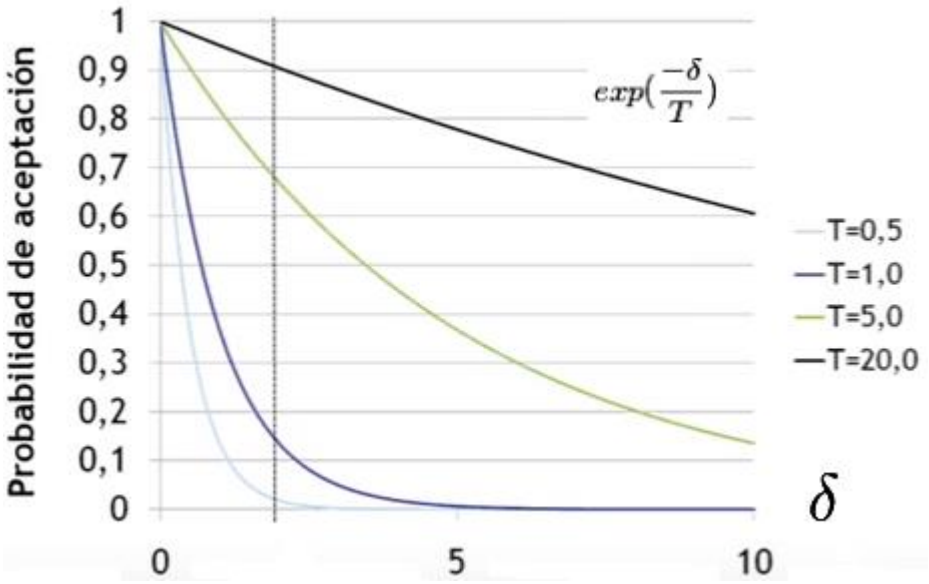


Figura 4.3. Gráfica de la probabilidad de aceptación para algoritmos de Simulated annealing

Analizando esta gráfica, se puede observar que para un delta determinado la probabilidad de aceptación es baja cuando la temperatura es baja y va creciendo según se incrementa la temperatura.

A continuación, se presenta una analogía entre los elementos de esta metaheurística presentes tanto en el campo de termodinámica como en la optimización combinatoria:

TERMODINÁMICA	OPTIMIZACIÓN COMBINATORIA
Configuración	Solución factible
Configuración fundamental	Óptimo
Estados de energía	Coste
Temperatura	Parámetro de control

Tabla 4.1. Analogía entre los elementos de la metaheurística Simulated annealing y la termodinámica

La ejecución de un algoritmo de Simulated annealing se realiza por iteraciones, cada uno de los cuales dura un número de ciclos determinado. En cada nueva iteración se parte con una temperatura menor a la de la iteración anterior, por lo que, en la primera iteración, la temperatura empieza tomando un valor máximo. Esto quiere decir que la probabilidad de aceptar una nueva solución candidata es alta durante las primeras iteraciones, aunque la diferencia de costes entre la de esta y la de la solución actual sea grande, lo que evita el estancamiento en óptimos locales.
Durante el transcurso de cada iteración, el coste de la función objetivo va disminuyendo (suponiendo que este sea el objetivo), pero también experimenta pequeñas subidas de vez en cuando. La temperatura de cada iteración se mantiene durante la duración de la misma, tras la cual disminuye, disminuyendo también la probabilidad de aceptación de nuevas soluciones candidatas.

Siguiendo esta dinámica, al final de la ejecución se alcanzaría una estabilización definitiva del coste, obteniéndose un óptimo global satisfactorio, siempre y cuando se haya configurado una buena estrategia de enfriamiento, pues mientras más lenta y adecuada sea ésta, mejores resultados nos reportará la performance del algoritmo.

El esquema general de un algoritmo de Simulated annealing es el siguiente:

```
Algoritmo Simulated annealing:
    Inicializar temperatura T
    Generar solución inicial

    Mientras no se alcance el número de iteraciones
    máximo OR otro criterio de parada, hacer
        Enfriar temperatura T
        Mientras no se alcance el número de ciclos
        de la iteración actual, hacer
            Generar nueva solución candidata
            modificando solución actual
            Si coste nueva solución candidata
            generada es mejor, entonces
                Se toma la nueva solución
                candidata como solución actual
            Si no
                Generar un cierto valor
                aleatorio p entre 0 y 1
                Si p <= probabilidad de
                aceptación, entonces
                    Se toma la nueva solución
                    candidata como solución
                    actual
                Fin si
            Fin si
        Fin mientras
    Fin mientras
Fin
```

4.4.4. GRASP

En las metaheurísticas presentadas hasta este punto, la aleatorización juega un papel importante. La aleatorización permite romper lazos, permitiendo explorar diferentes trayectorias en la construcción de una solución. También permite mostrar diferentes partes de grandes vecindarios en el espacio de soluciones.

Los algoritmos de la metaheurística GRASP (Greedy Randomized Adaptive Search Procedure) presentan también características de aleatorización en su estructura. Además, adoptan algunos comportamientos de los algoritmos voraces, por lo que existe un debate sobre si considerar a GRASP como realmente una metaheuristica.

Los algoritmos GRASP generan trayectorias para obtener un óptimo. Son fáciles de implementar y su ejecución se realiza en ciclos, denominados ciclos GRASP. Cada uno de estos ciclos consta de tres fases:

- *Fase de preprocesamiento*: En esta fase se buscan esquemas que sean buenos para construir buenas soluciones al problema. Además, se genera una solución inicial.

- *Fase de construcción*: Partiendo de la solución inicial de la fase de pre procesamiento, mediante iteraciones se avanza de manera voraz y aleatoria hasta un límite de factibilidad. Durante esta fase, cada solución candidata en construcción estará formada únicamente por aquellos elementos que pueden ser incorporados sin que se pierda por ello factibilidad. La selección del siguiente elemento a ser incorporado a una solución candidata se lleva a cabo de acuerdo a una función de evaluación voraz.
  Se tiene una lista reducida de candidatos (RLC), donde se almacenan los n mejores elementos candidatos que pueden llegar a formar parte de una solución candidata. Se selecciona un elemento de dicha lista de manera aleatoria. Esta selección aleatoria de candidatos es lo que hace realmente potente a GRASP, puesto que, al no elegir siempre al mejor, el algoritmo no camina directamente hacia el límite de factibilidad, si no que genera muchas más trayectorias aleatorias mientras avanza hacia él, permitiendo explorar más partes del espacio de soluciones.

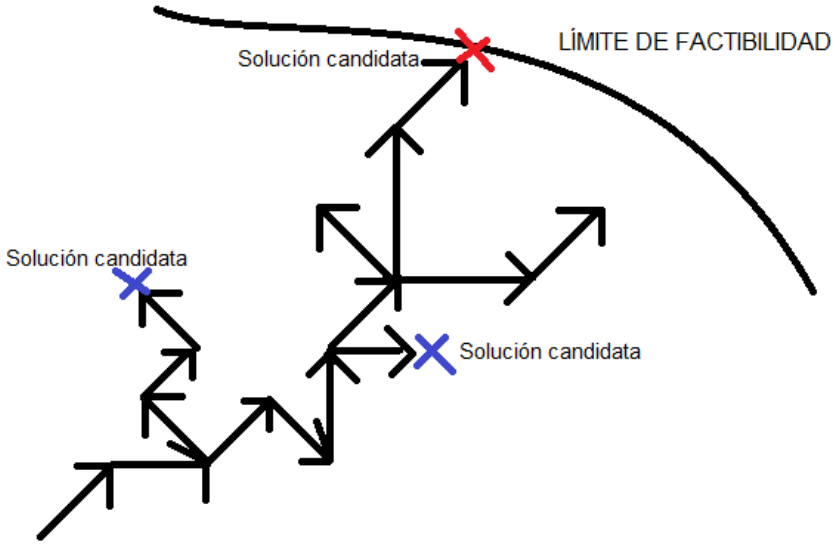


Figura 4.4. Trayectorias generadas por algoritmos de la metaheurística GRASP

- *Fase de búsqueda*: después de haber alcanzado una solución candidata en el límite factibilidad, se puede realizar una búsqueda en vecindad para intentar mejorar la calidad de la solución obtenida hasta el momento.

La solución final será la mejor obtenida después de realizar todos los ciclos.

El esquema general de un algoritmo GRASP es el siguiente:

```
Algoritmo GRASP:
    Mientras no se hayan completado todos los ciclos GRASP, hacer
        Inicializar conjunto C elementos candidatos a formar una solución
        Inicializar lista reducida de candidatos (RLC)
        Mientras no se haya alcanzado el límite de factibilidad, hacer
            Mientras no se alcance el número de elementos de C AND no se alcance la capacidad de RLC, hacer
                Seleccionar un elemento e de C de acuerdo a una cierta función de selección voraz
                Añadir el elemento e seleccionado a RLC
                Actualizar C
            Fin mientras
            Seleccionar aleatoriamente un elemento e de RLC
            Incorporar e a la solución actual
            Reevaluar costes de la solución candidata actual
            Actualizar RLC
            Realizar una búsqueda en vecindad para mejorar la calidad de la solución candidata actual obtenida
        Fin mientras
    Fin mientras
    Devolver la mejor solución obtenida
Fin
```

"Si amas la vida, no pierdas el tiempo, de tiempo está hecha la vida", Bruce Lee.

Capítulo 5

Proyecto

En este capítulo se va a presentar la gestión del proyecto, mediante el plan de trabajo y la planificación temporal que se siguieron para lograr la entrega del mismo.

5.1. Plan de trabajo

Para asegurar la conclusión del desarrollo del proyecto de este Trabajo de Fin de Grado, de manera que podamos realizar su entrega y presentación dentro de los plazos establecidos, se realizaron las siguientes actividades:

Actividad	Involucrados	Riesgo principal
Reunión inicial	Equipo de trabajo y tutor a cargo	Los involucrados no logran acordar una fecha para la reunión
Organización del proyecto y división del mismo	Equipo de trabajo	No es posible llegar a un acuerdo sobre la organización del plan de trabajo a seguir
Reuniones y acuerdos con el cliente	Equipo de trabajo y resto de involucrados (tutor a cargo)	Falta de claridad en cuanto al avance del proyecto y las especificaciones que el cliente desea
Desarrollo de un plan de riesgos	Equipo de trabajo	No se tienen en cuenta las especificaciones que el cliente quiere para el proyecto
Plan de calidad	Equipo de trabajo y tutor a cargo	El desarrollo del proyecto no cumple con las especificaciones
Planificación temporal	Equipo de trabajo y tutor a cargo	No se cumplen los tiempos establecidos
Revisión final	Equipo de trabajo y tutor a cargo	El resultado final no es lo que el cliente desea

Tabla 5.1. Plan de trabajo del proyecto

5.2. Planificación temporal

El desarrollo de nuestro proyecto fue dividido entre ambos miembros de este equipo de trabajo. Teniendo en cuenta el trabajo realizado y nuestras estimaciones, definimos la siguiente tabla que plasma las distintas fases por las que hemos pasado:

Fase	Fechas
Inicio	Inicio: 28 de septiembre de 2015 Fin: 31 de octubre de 2015
Elaboración	Inicio: 1 de noviembre de 2015 Fin: 31 de diciembre 2015
Construcción	Inicio: 1 de enero de 2016 Fin: 30 de abril de 2016
Transición	Inicio: 1 de mayo de 2016 Fin: 30 de junio de 2016
Entrega	1 de septiembre de 2016

Tabla 5.2. Planificación temporal del proyecto

Cuando realizamos esta estimación de fechas, nos basamos en experiencias anteriores de proyectos de similar complejidad. Como es lógico, hubo fases, las cuales o bien las cumplimos a tiempo o bien nos retrasamos. Pero hay que decir que estos retrasos no supusieron graves consecuencias en el desarrollo. Se proyectaron una serie de días para realizar revisiones o cubrir retrasos, y para asegurar todo lo posible la corrección del proyecto.

Hitos de cada fase

A continuación, pasamos a analizar los hitos necesarios para conseguir completar cada fase y avanzar a la siguiente:

Fase	Hito
Inicio	Se especificará la visión del proyecto y la idea inicial para el desarrollo de la aplicación. Se llegará a un acuerdo sobre los problemas de optimización a resolver y se realizará un estudio de estos. También se seleccionarán los algoritmos para resolver los problemas y se empezarán a estudiar. Se identificarán los principales casos de uso de la aplicación. Se identificarán los riesgos importantes del desarrollo.
Construcción	Se construirá la aplicación de acuerdo a los requisitos especificados
Transición	Se realizarán test en busca de fallos y se repararán
Entrega	Se entregará la aplicación junto con el documento de la memoria de desarrollo

Tabla 5.3. Hitos de cada fase del proyecto

5.3. Gestión del Control de Versiones

Para la gestión de documentos hemos usado Google Drive y Dropbox. Gracias a estos gestores, sobre todo a Google Drive, en concreto a su servicio Google Docs, pudimos regular los cambios gracias al historial de cambios, además, pudimos trabajar de forma paralela en un mismo documento, y llevar un control de versiones.

Para la gestión de versiones de los ficheros de código fuente de la aplicación, hemos usado GitHub, y para ello hemos creado un repositorio privado. Como capa superior a este servicio, hemos utilizado la herramienta SourceTree como cliente de escritorio de Git.

SourceTree es una aplicación muy potente y muy útil con la que gestionar los cambios, las ramas, las mezclas de código, las subidas, etc. Puede trabajar con repositorios de código fuente de GitHub, Bitbucket, Subversion, ...Además, dispone de una simple interfaz gráfica que organiza los cambios en forma de árbol, permitiendo visualizarlos cómodamente. Esta herramienta advierte de los cambios producidos en otras ramas, o en la rama maestra.

"La imaginación es más importante que el conocimiento. El conocimiento es limitado, mientras que la imaginación no", Albert Einstein.

Capítulo 6

Adaptación e implementación

En este capítulo presentaremos las estructuras de datos que hemos implementado y que nos han servido para modelar los problemas resueltos, presentaremos algunas heurísticas que nos ayudaron a obtener soluciones iniciales de los problemas con las que pudimos empezar a optimizar y explicaremos cómo hemos adaptado las metaheurísticas estudiadas a estos problemas y cómo las hemos implementado.

6.1 Estructuras de datos

6.1.1. Estructuras de datos para el TSP

6.1.1.1. Mapas

Para modelizar el problema del TSP hemos definido unas estructuras llamadas *mapas,* cuya función principal es ofrecer la *distancia* que existe entre ciudades. Además, estas estructuras cuentan con métodos para saber el número de ciudades total, saber si se puede ir de forma directa de una ciudad a otra, la distancia entre un par de ciudades, etc.

Hemos desarrollado diferentes tipos de mapas mediante diversos tipos de implementaciones. La implementación más sencilla para un mapa es la realizada mediante una *matriz de distancias*, con la que se puede conseguir la distancia entre dos ciudades accediendo a las posiciones correspondientes en dicha matriz. También se puede saber si existe conexión entre un par de ciudades cualesquiera. Si esta esta distancia tiene un valor prohibido como Infinito, es porque no existe conexión.

Además, los datos de entrada necesarios para la construcción de mapas pueden ser obtenidos de varias fuentes como:

- *Ficheros de texto*, cada uno de los cuales es *parseado* para interpretar el contenido de sus líneas. Estas líneas pueden definir o coordenadas o distancias.

- *Peticiones HTTP* para el uso del servicio de cálculo de distancias entre dos puntos ofrecido por Google Maps. Gracias a esto, podemos crear un mapa con ciudades y distancias reales según el medio de transporte utilizado.

A continuación, presentamos algunos tipos de mapas implementados:

6.1.1.1.1. Mapa Ciudades

Hemos definido un mapa que contiene unos elementos llamados *Ciudades*. Una ciudad contiene información sobre esta, como su id, nombre, …
Un mapa de ciudades tiene un método que devuelve la distancia de una ciudad a otra ciudad.

De este tipo de mapa, hemos creado diversas implementaciones al igual que diferentes formas de representar una ciudad, entre las cuales, por ejemplo, está el hecho de guardar las *coordenadas,* gracias a las cuales y por medio de una sencilla operación matemática, podemos calcular la distancia a otra ciudad.

Otra forma de representar ciudades ha sido la de incluir en cada una de ellas una *lista de ciudades adyacentes* con sus respectivas distancias, de manera que se pueden recorrer los K vecinos de cada ciudad.
Las distancias se calculan y se almacena una única vez, pudiendo luego obtenerse mediante accesos constantes. Esto es útil puesto que, aunque partimos de ejemplos de ciudades de 2 dimensiones, podríamos tener z-dimensiones, lo que aumentaría el coste de la operación de calcular la distancia.

Hemos hablado de *distancia*, pero no la hemos definido, y esto es debido a que hemos separado también el cálculo de la distancia de las estructuras en sí mismas. La distancia que solemos usar es la *Euclídea* de 2 dimensiones, pero también existe la distancia Euclídea de n dimensiones, y otros tipos como la distancia Manhattan.

Aunque para la ejecución de pruebas de los algoritmos para resolver el TSP, podemos usar distancias inventadas entre ciudades de mapas inventados, estas distancias pueden ser calculadas sobre mapas del mundo real, como cuando hacemos uso de la API de Google Maps.

6.1.1.1.2. Mapas Grafo

Aunque tenemos muchas implementaciones diferentes de mapas, pensamos en que quizá necesitaríamos algunas otras, por lo que hemos implementado también un tipo de mapa basado en un grafo valorado. Dentro de los grafos valorados, existen diversas implementaciones de los mismos, ya sea por matriz de distancias, o por lista de adyacentes. Utilizar una u otra estructura puede crear una gran diferencia. Si utilizamos lista de adyacentes, y, por ejemplo, una ciudad solo tiene un vecino, este estará contenido en una lista de un solo elemento, mientras que, si usamos matriz de distancias, el número de vecinos de cada ciudad será siempre N, lo que supone espacio desperdiciado en memoria.

6.1.1.1.3. Mapas Híbridos

Debido a que todas las formas de implementar los mapas que hemos nombrado son útiles

dependiendo de para qué, hemos creado también *hibridaciones* entre ellas para poder obtener lo bueno de todas, aunque esto pueda aumentar el coste de construcción. Entre estas hibridaciones podemos encontrar, *matriz con lista de ciudades*, las cuales ya no tienen por qué almacenar distancias. Otra ventaja de estas hibridaciones es el hecho de que, según diferentes datos de entrada, se puede crear un mismo tipo de mapa que englobe varias necesidades.

6.1.1.2. Solución del problema

La solución a una instancia del problema del TSP está representada mediante un objeto solución. Este objeto cuenta con dos atributos:

- *Una permutación circular de ciudades que representa un ciclo Hamiltoniano*. Las ciudades de este ciclo están contenidas dentro de un vector y ordenadas según sean recorridas.

- *El coste total de la ruta de la solución*. Este coste puede ser medido por factores como la distancia, el tiempo, etc.

6.1.2. Estructuras de datos para el problema de la Mochila 0-1

6.1.2.1. Mochilas

En la modelización del problema de la Mochila 0-1 se han definido e implementado varios tipos de mochila, concretamente dos, las cuales pasamos a presentar a continuación.

6.1.2.1.1. Mochila básica

La mochila básica es una estructura que, entre otras operaciones, posee las de obtener su capacidad total u obtener el beneficio y el coste de un determinado ítem a través de su índice.

Dicho esto, desde nuestro punto de vista, esta es una estructura básica, aunque poco útil, la cual contiene listas de costes y de beneficios de los ítems. El coste y el beneficio son valores parametrizables, por lo que, por ejemplo, el coste podría ser el peso y el beneficio podría ser dinero.

6.1.2.1.2. Mochila de ítems

Para mejorar la estructura de la mochila básica anterior, se encapsulan los costes y beneficios dentro de un nuevo tipo de objeto conocido como *ítem*.

Un ítem, además de un coste y un beneficio, cuenta con id que lo identifica de manera unívoca y con una posición dentro de la mochila. Posee operaciones para comparar un ítem con cualquier otro, ya sea por coste o por beneficio.

Este tipo de mochila tiene entre otros métodos, el de ordenar ítems o el de obtener el índice

de cualquiera de ellos dentro de una mochila ordenada. Para este último método, o se accede a una variable de posición que posee cada ítem en el caso de que sea una mochila única, o se accede a un mapa clave-valor, el cual por cada ítem almacena su posición.

Para comparar los ítems, se han creado varios comparadores: por coste, por beneficio, por id, y por ratio beneficio/coste.

6.1.2.2. Solución del problema

La solución al problema de la Mochila 0-1 está representada mediante un objeto que encapsula las propiedades de dicha solución. Estas propiedades son las siguientes:

- Se tiene una lista de ítems seleccionados por el algoritmo que ha resuelto el problema.

- Además de la lista de ítems anterior, se tiene una lista de booleanos de ítems seleccionados.

- Se tiene el coste y el beneficio de una determinada solución.

Un pequeño problema de parametrizar los ítems, es que no se pueden sumar porque los operadores no están sobrecargados.

6.2. Heurísticas para el TSP

6.2.1. Heurística del vecino más cercano

El comportamiento de este método heurístico voraz se basa en buscar la ciudad más cercana a la actual en el recorrido. Estando en la ciudad actual, se busca la ciudad vecina más cercana, de entre las no recorridas aún. Al avanzar hacia la ciudad elegida, la marcamos como recorrida para descartarla en búsquedas posteriores y la consideramos como ciudad actual del recorrido.

Se hace un recorrido de todas las ciudades, por lo que tiene coste lineal, pero para mejorarlo. El problema reside en que estando en la ciudad actual, hay probar todas las posibilidades para seleccionar la siguiente ciudad a la que saltar, provocando que se el coste aumente.

6.2.2. Heurística de inserción

Este método se basa en dividir las ciudades en 2 conjuntos: *utilizadas* y *no utilizadas*. Es similar al algoritmo de Prim que busca un árbol *recubridor* mínimo (MST o Minimum Spanning Tree) en un grafo. Al igual que en este, se tiene una variable que sirve para saber la distancia entre cada ciudad del conjunto de utilizadas a otra ciudad del conjunto de no utilizadas. Esta distancia sirve para saber cuál meter a continuación como parte de la solución. Una vez insertada la nueva ciudad en la ruta, se actualizan las distancias.

6.2.3. Optimizaciones

Las heurísticas anteriores pueden ser mejoradas para intentar obtener soluciones de mayor calidad. De entre las alternativas que existen para ello, hemos utilizado la optimización denominada 2-Opt.

2-Opt rompe en dos una ruta que se cruza en un punto y la vuelve a reconstruir intercambiando los orígenes y destinos de estos puntos**[15]**. Explicado con ejemplos consiste en: si tenemos dentro del recorrido unidas las ciudades b con e y c con f, consiste en unir b con c y e con f.

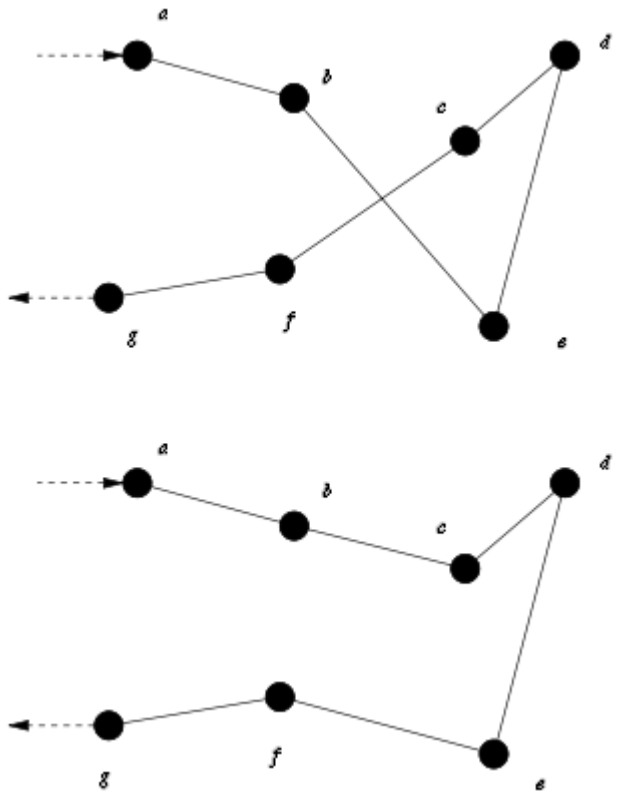


Figura 6.1. Optimización 2-opt. Recuperada de https://en.wikipedia.org/wiki/2-opt#/media/File:2-opt_wiki.svg

Si después de reconstruir la ruta, su nueva distancia total es menor que la de la ruta original, se toma a la nueva ruta como la mejor actual.

En el modelo de problema que estamos usando, no existen restricciones de factibilidad en cuanto a si podemos o no visitar cada una de las ciudades, y dado el caso de no poder, el coste la arista (i, j) que une un par de ciudades incomunicadas lo representaríamos por un valor infinito. Esto quiere decir que la metaheurística podría tomar aristas con valores infinitos, situación que pensamos que se podría arreglar por medio de reconstrucciones 2-Opt.

6.3. Metaheurísticas para el TSP

6.3.1. Adaptación e implementación de algoritmos genéticos

Para la codificación de los individuos (soluciones), hemos usados una permutación de valores enteros, por lo que la factibilidad del individuo depende de si es una permutación. Dicho de otra forma, si tenemos n genes de valores entre 1 y n, y sin repetir, cada gen representa una ciudad. Para calcular el fitness, se puede usar la distancia total del individuo, es decir, tomando cada gen como ciudad, y sumando las distancias entre ellas.

Para la selección de individuos, hemos probado con los nombrados, *Ruleta* y *Torneo*. El más efectivo ha sido un *Torneo* de tamaño 5. Esto ha sido igual en el problema de la Mochila 0-1.

Debido a la restricción de que una solución es una permutación de ciudades, tuvimos que utilizar operadores de cruce propios para permutaciones tales como: *PMX (Partially Mapped Crossover)*, C*X (Cycle Crossover), EX (Edge Crossover)*.

Como operadores de mutación también tuvimos que utilizar los especializados en permutaciones, como, por ejemplo, los de intercambio de posiciones como pueden ser: *IVM o inversión*, *DM o mezcla*: *SM o swap*.

Hemos hecho pruebas con los operadores anteriores y tanto los cruzadores como los mutadores ofrecen una eficacia similar, según se modifiquen parámetros tales como el tamaño de la población y el número de generaciones.

Hemos implementado verificadores y correctores para asegurarnos de que un individuo sea factible, aunque usando los operadores mencionados, no son necesarios puesto que a veces van incluidos dentro de estos.

Para optimizar, utilizamos el algoritmo 2-opt encapsulado en un optimizador

6.3.2 Adaptación e implementación de la metaheurística ACO

ACO se adapta de forma natural a la resolución del TSP, puesto que esta está diseñada para trabajar con caminos en grafos, tales como caminos mínimos o ciclos como en este caso. A su vez, hemos estudiado dos tipos de implementación ACO, una más estándar y otra elitista, a la que hemos denominado ACE. La implementación estándar, se basa en el modelo básico de optimización de colonia de hormigas en el cual, cada hormiga deposita las feromonas correspondientes a su solución (un número de feromonas proporcional a la *optimalidad* de la solución que encapsulan), y en cada iteración existe una evaporación de estas. Sin embargo, en la versión elitista, solo la hormiga con la mejor solución conseguida en la iteración, es la que deposita feromonas. En el problema del TSP, con las instancias del problema que hemos probado y sin utilizar una solución inicial, ambas versiones han ofrecido una eficacia similar. Sin embargo, la versión elitista, ha conseguido tiempos menores, como se muestra más adelante en el capítulo 7 sobre comparativas.

Después de esto, se ha optimizado en algoritmo ACE, con una búsqueda, lo que ha reducido de forma considerable el tiempo. Dicha búsqueda local, se puede realizar en la hormiga mejor, o en cada una. Al hacerlo en cada una, el tiempo vuelve a aumentar.

6.3.3. Adaptación e implementación de la metaheurística Simulated Annealing

Para adaptar la metaheurística de Simulated annealing al TSP, tuvimos que analizar los elementos del problema para poder realizar una analogía entre estos y los propios de la metaheurística.

Nos preguntamos cómo asemejar los diferentes estados de la materia que se obtienen por el reordenamiento de sus partículas debido a los cambios de temperatura. A lo que decidimos representar cada estado como una permutación de ciudades, la cual forma una ruta candidata. El vecindario de estados (rutas candidatas) derivado de cada estado parcial obtenido, estaría formado por el conjunto de nuevas configuraciones de los mismos, obtenidas al volver a reordenar sus partículas (ciudades).

El reordenamiento de la materia se lleva a cabo mediante los movimientos de sus partículas. En los algoritmos de Simulated annealing, estos movimientos suelen dar lugar a alteraciones mínimas en el último estado. Cuando estas alteraciones son mínimas, se preservan las mejores configuraciones, cambiando solamente las peores partes de estas. Traducido al problema del TSP, estas partes son las ciudades de una ruta. Intercambiando ciudades de una ruta, también cambiamos conexiones, y como consecuencia se obtienen mejores o peores distancias.

En cuanto a la *distribución de Boltzman* $exp(\frac{-\delta}{T})$, la cual mide la probabilidad de aceptar nuevas rutas candidatas, podemos tomar δ como la distancia de la ruta candidata en cuestión.

En un principio implementamos el algoritmo tomando como solución de partida una muy básica, que consistía en una permutación de ciudades consecutivas una a una, desde la ciudad 0 a la n-1, siendo n el número total de ciudades. Utilizar este tipo de solución de partida en ocasiones no proporcionaba otras soluciones de muy buena calidad.
Para mejorar esta situación, decidimos obtener una solución inicial mediante la heurística del vecino más cercano.

6.3.4. Adaptación e implementación de la metaheurística GRASP

Como hemos explicado, Los métodos GRASP podemos dividirlos en tres fases, sin embargo, para el TSP hemos decidido implementar únicamente la fase de construcción y la fase de búsqueda:

Fase de construcción: Creamos una solución aleatoria partiendo de una ciudad inicial. Dicha ciudad inicial la retiramos del conjunto de candidatos. A continuación, y hasta quedarnos sin ciudades candidatas, escogemos cualquiera de forma aleatoria dentro de la lista de ciudades candidatas. Según vamos eligiendo y metiendo ciudades en nuestra solución, las vamos descartando. De esta forma se consigue conservar la factibilidad al no repetir ciudades.

Fase de búsqueda: una vez construida esta solución inicial, buscamos en su región de vecindad, una solución mejor. Para ello, primero, y siempre que sea posible hacemos intercambios en la permutación de ciudades que representa la solución actual con el algoritmo 2-opt.

Por último, y teniendo en cuenta que esta técnica no supone en este caso demasiado aumento en la complejidad temporal de la metaheurística, hemos optado por hacer una búsqueda tipo *best improvement*, en vez de *first improvement*, puesto que creemos que es más útil mejorar la solución conseguida comparando con todas las candidatas del vecindario, que parar al encontrar la primera que la mejore.

Para el problema del TSP, el límite de factibilidad hacia el que avanza la metaheurística, no lo podemos definir tan fácilmente, puesto que en este caso dicha factibilidad se basa en la permutabilidad de una secuencia de ciudades. Por lo que para ello, la forma más fácil de llenar una ristra de n ciudades que sean una permutación es que al considerar la siguiente candidata, esta sea una ciudad perteneciente a la lista de candidatas no utilizadas aún, por lo que no se repiten ciudades en la solución, y además, hay que tener en cuenta que la longitud de la permutación es igual que la de la lista de ciudades del problema y no existe ninguna otra forma de conseguir llenarla de

manera correcta que usando solamente las ciudades de esta lista.

Para optimizar, se ha considerado utilizar una solución inicial, la cual se compone de una serie de aristas. A continuación, se va creando una solución inicial de forma aleatoria, pero permitiendo insertar dichas aristas.

6.4. Metaheurísticas para el problema de la Mochila 0-1

6.4.1. Adaptación e implementación de algoritmos genéticos

En este problema los genes de los individuos (soluciones) representan un valor booleano que significa si se elige el ítem de la posición del gen. La restricción que se añade, es que la suma (sumatorio) de los costes de los ítems elegidos, sea menor o igual que la capacidad de la mochila. Como fitness usamos la suma (sumatorio) de los beneficios de los ítems elegidos.
Además, se puede utilizar una penalización por no factibilidad, respecto a la diferencia entre la suma de los costes y la capacidad de la mochila, cuando esta es superada.

Al igual que en TSP, para la selección de individuos, hemos probado con el *método de la Ruleta* y el *método del Torneo*, y el más efectivo ha sido un Torneo de tamaño 5.

Como la única restricción que se tiene es sobre la capacidad de la mochila, podemos usar operadores de cruce básicos tales como los basados en cortes como, SPX (Single Point Crossover), UPX (Uniform Point Crossover).

Como operadores de mutación tampoco tenemos que usar especializados y podremos usar por ejemplo el más básico, el operador *flip*, que, dado un ítem, si este pertenece a la solución, lo sacamos, y si no está, lo metemos, o el operador de intercambio de posiciones, que elige 2 ítems al azar y si uno está dentro y otro fuera de la solución, saca al que está dentro y mete al que estaba fuera, aunque esto puede generar un individuo (solución) no factible.

Hemos hecho pruebas con los operadores anteriores y tanto los cruzadores como los mutadores ofrecen una eficacia similar, según se modifiquen parámetros tales como el tamaño de la población y el número de generaciones.

Para verificar de manera rápida la factibilidad creamos un verificador que comprueba si el coste acumulado de los ítems de la solución, supera a la capacidad de la mochila.

Para mejorar los individuos (soluciones), usamos la búsqueda local basada en el intercambio.

6.4.2. Adaptación e implementación de la metaheurística ACO

Una vez implementada esta heurística para el TSP, la hemos adaptado al problema de la Mochila 0-1. Para ello hemos escogido la alternativa elitista, creyendo que, puesto que mejora los resultados en el problema TSP, podría hacer lo mismo con este problema.

Para adaptar ACO al problema de Mochila 0-1, manejamos dos alternativas:

- Inspirándonos en su adaptación para el TSP, de manera que se crea un mapa donde las ciudades son los ítems, y donde existen aristas que conectan cada ítem con todos los demás, formando un grafo completo.

- La segunda alternativa consiste en unir los ítems ordenados por medio de unos enlaces que indican si hemos elegido el ítem i-ésimo. A continuación, se muestra una figura con esta estructura:

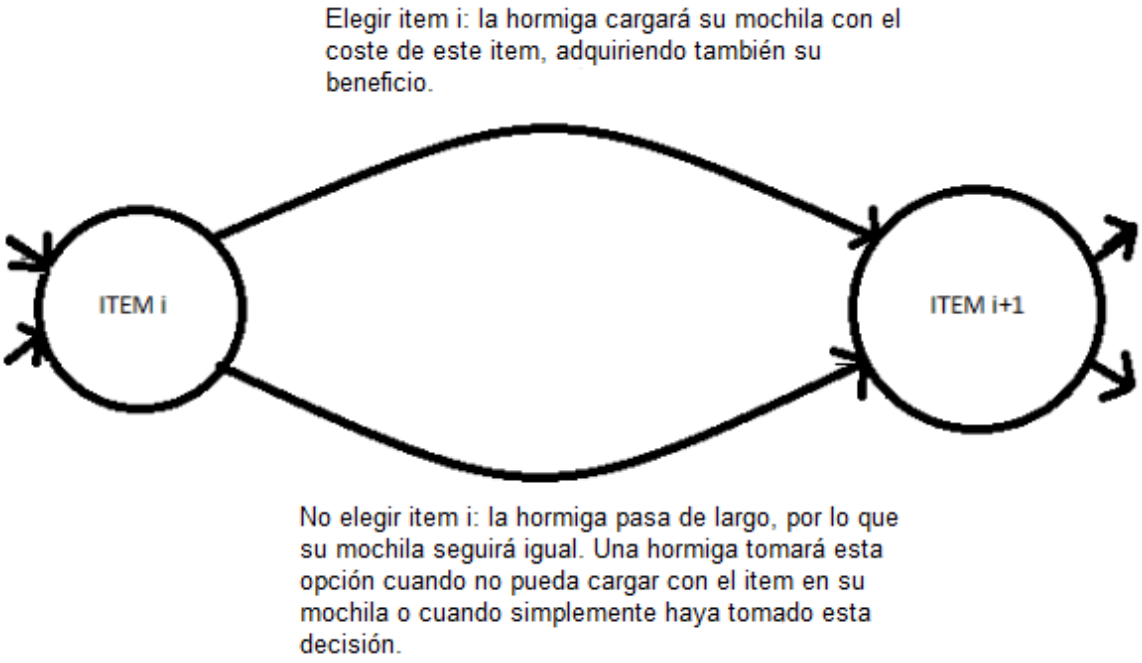


Figura 6.2. Elección de arista al momento de elegir o no un ítem

Con la primera alternativa, se podría reutilizar el algoritmo implementado para el TSP, aunque esta es una adaptación un poco forzada.

La segunda alternativa es la utilizada y es la mejor, aunque haya que implementar un algoritmo nuevo modificando el algoritmo implementado para el TSP. Dicho algoritmo se verá alterado en los siguientes aspectos:

- El mapa se verá modificado, y concretamente, la estructura que se crea será un grafo en el que cada nodo es un ítem, y los enlaces son dobles, conectando cada ítem i con el ítem i+1. Son dobles porque representan el camino de elegir un ítem o no.

- La información de la hormiga se modifica, y en vez de almacenar solo el coste, almacena también el beneficio. Pero con la particularidad de que es el beneficio el que se usa como criterio de *optimalidad*, por lo que el problema pasa a ser de *máximos*, en vez de *mínimos* (en el TSP se busca la ruta de coste mínimo). El coste en este caso será usado para saber el llenado de la mochila.

Al llegar a un ítem, se comprueba si se puede elegir, y en caso de no poderse, la hormiga estará obligada a ir por la rama de no elegir. Si se puede, la hormiga podrá optar por ambas, de forma aleatoria y usando las feromonas de ambas ramas.

Al igual que con el TSP, se ha optimizado en algoritmo ACE, con una búsqueda, lo que ha reducido de forma considerable el tiempo. Dicha búsqueda local, se puede realizar en la hormiga mejor, o en cada una. Al hacerlo en cada una, el tiempo vuelve a aumentar.

6.4.3. Adaptación e implementación de la metaheurística Simulated Annealing

La manera de proceder para adaptar Simulated annealing al problema de la Mochila 0-1 fue la misma que para el TSP, es decir, realizamos una analogía entre los elementos del problema respecto a los de la heurística:

Los diferentes estados de la materia fueron representados mediante permutaciones de ítems, las cuales forman configuraciones (soluciones) de la mochila, siendo el vecindario de estados de cada una de estas configuraciones, el conjunto de configuraciones de la mochila que derivan de cada una de ellas.

Las partes eliminadas y sustituidas de cada configuración de la materia, vendrían a ser los ítems de cada configuración de la mochila. Cuando se sustituyen estas partes se obtienen mejores o peores resultados en cuanto a peso total ocupado de la mochila.

En cuanto a la *distribución de Boltzman* $exp(\frac{-\delta}{T})$, la cual, en este caso, mide la probabilidad de aceptar nuevas configuraciones candidatas de la mochila, podemos tomar δ como el peso total que ocupa cada una de estas configuraciones en la mochila del problema.

En este caso, hemos partido de una solución inicial obtenida mediante la heurística voraz de ratio beneficio/costo.

6.4.4. Adaptación e implementación de la metaheurística GRASP

Cuando nos pusimos a adaptar esta metaheurística al problema de la Mochila 0-1, empezamos por buscar algún esquema para insertar en las soluciones candidatas siempre los mejores ítems posibles, con el objetivo de obtener soluciones de alta calidad. Ante esto, decidimos crear una lista de ítems candidatos ordenada de manera decreciente según un índice de sensibilidad beneficio/costo.

Pasando a la fase constructiva de la metaheurística, decidimos establecer un límite de factibilidad tomando para ello a la capacidad de la mochila del problema. Por otro lado, se utilizó una lista reducida de ítems candidatos donde almacenar cierto número de ítems (nosotros decidimos almacenar un máximo de 5 ítems) de la lista de candidatos anterior, para lo que establecimos el siguiente filtro de selección de ítems recomendado en la literatura:

$$\frac{ben_i}{cost_i} \geq \left(\frac{ben_i}{cost_i}\right)_{max} - \alpha \left\{ \left(\frac{ben_i}{cost_i}\right)_{max} - \left(\frac{ben_i}{cost_i}\right)_{min} \right\}$$

En esta lista reducida se van introduciendo y seleccionando aleatoriamente ítems y se introducen en la mochila hasta alcanzar el límite de factibilidad establecido.

Como ya se explicó en el capítulo 4, los algoritmos GRASP avanzan hasta el límite de factibilidad generando trayectorias aleatorias en búsqueda de soluciones de alta calidad. Pensando en qué hacer para mejorar aún más si cabe la mejor solución obtenida hasta el momento una vez alcanzado dicho límite, nos hemos decantado por hacer una búsqueda en vecindad de tipo best improvement, que consiste en analizar todas las soluciones candidatas del vecindario, y quedarse con la mejor de todas que supere a la solución actual, si es que existiera. En el caso del problema de la Mochila 0-1, esto ha consistido en seguir seleccionando ítems según beneficio/costo de entre los restantes de la lista de ítems candidatos ordenada generada al principio.

"Defiende tu derecho a pensar, incluso pensar de manera errónea es mejor que no pensar", Hipatia de Alejandría.

Capítulo 7

Comparativas

En este capítulo se van a presentar comparativas entre los algoritmos implementados para cada problema elegido, siguiendo diferentes criterios como el tiempo utilizado o la calidad de las soluciones obtenidas.

Para realizar estas comparativas hemos implementado clases de prueba, las cuales siguen el siguiente funcionamiento:

- Se insertan en una lista todos los algoritmos parametrizados.
- Se ejecutan y se almacenan los resultados en una estructura.
- Si es necesario, se ordenan y se busca el mejor para compararlo.
- Se guardan en fichero, según problema y según algoritmo. Para ello se crean subcarpetas.

Las comparativas se realizarán con tiempos menores a 1 minuto. En problemas complejos y/o grandes es posible que sea interesante dejar ejecuciones más largas (horas o incluso días) para obtener buenas soluciones.

Los datos de las instancias de los problemas son almacenados en ficheros, con extensión .tsp para el TSP y .txt para el problema de la Mochila 0-1.
Como resultado de la ejecución de las pruebas, obtenemos unos ficheros .csv (Comma Separated Value), un formato que de forma sencilla almacena cada valor de un dato compuesto en columnas, representando cada fila dicho dato. Cada fichero, para una mejor comprensión, tiene cabecera con el mismo formato. Ej:
algoritmo;eficacia;
GA;100;

A continuación, presentamos una leyenda con los nombres utilizados para cada algoritmo durante todas las comparativas y experimentos realizados:

Algoritmos exactos:

- BT (Backtracking).
- BTOrd (Backtracking con ítems ordenados según su coste).
- PD (Programación dinámica).

Heurísticas:

- vecino (Vecino más cercano).
- INCP Opt (Heurística optimizada de inserción posterior, implementada con conjuntos).
- INSAP Opt (Heurística optimizada de inserción posterior, implementada con arrays).

Metaheurísticas:

Simulated annealing:

- sim (Algoritmo de enfriamiento simulado).
- simi (Algoritmo de enfriamiento simulado, con solución inicial de partida).

GRASP:

- GRASP (Procedimiento aleatorio voraz de búsqueda adaptativa).
- GRASPi (Procedimiento aleatorio voraz de búsqueda adaptativa, con solución inicial de partida).

Algoritmos genéticos:

- MAG (Algoritmo memético generacional).
- MAGEsp (Algoritmo memético generacional específico).
- GAG (Algoritmo genético generacional).
- GAGEsp (Algoritmo genético generacional específico).

Metaheurísticas inspiradas en colonias de hormigas:

- ACE Opt (Metaheurística optimizada, inspirada en colonias de hormigas elitistas).
- ACE OptTot (Metaheurística optimizada, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución).
- ACEi Opt (Metaheurística optimizada, con solución inicial de partida, inspirada

en colonias de hormigas elitistas).
- ACEi OptTot (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución).

También incluimos a continuación los parámetros de cada algoritmo utilizado para resolver los problemas y sus respectivos valores:

Tanto para el TSP como para el problema de la Mochila 0-1, los valores iniciales de los parámetros utilizados, son los siguientes:

- En los algoritmos ACEi Opt y ACEi OptTot:

  BETA = 2
  GAMMA = 0.1
  q0 = 0.9
  Q = 1.0
  NUM_HORMIGAS = 2
  ITER_MAX = 50000

- En los algoritmos GAG, GAGEsp, MAG y MAGEsp:

  TASA_MUTACIÓN = 0.015
  TASA_CRUCE = 0.8

  ELITISMO = true
  MEMETISMO = true (MAG y MAGEsp), false (GAG y GAGEsp)

  GENERACIONES = 200
  TAM_POBLACIÓN = 50

- En los algoritmos GRASP y GRASPi:

  ALFA = 0.55
  NUM_ITERACIONES = 10

- En los algoritmos sim y simi:

  TEMPERATURA_INICIAL = 100
  TEMPERATURA_FINAL = 0.1
  NUM_ITERACIONES = 1000
  CICLOS DE BÚSQUEDA LOCAL = 100

  Sin embargo, a la hora de realizar las pruebas fueron modificados de la siguiente manera:

- **En el TSP:**

Para las pruebas con los algoritmos ACEi Opt y ACEi OptTot se han modificado los valores de los siguientes parámetros:

NUM_HORMIGAS = 2
ITER_MAX = 100

Para las pruebas con los algoritmos GAG y GAGEsp se han modificado los valores de los siguientes parámetros:

TAM_POBLACIÓN = 10;

Para las pruebas con los algoritmos MAG y MAGEsp se han modificado los valores de los siguientes parámetros:

GENERACIONES = 20

Para las pruebas con los algoritmos GRASP y GRASPi, se han modificado los valores de los siguientes parámetros:

ALFA = 1000
NUM_ITERACIONES = 1000

Para las pruebas con los algoritmos sim y simi, se mantienen los valores iniciales de sus parámetros

- **En el problema de la Mochila 0-1:**

Para las pruebas con los algoritmos ACEi Opt y ACEi OptTot se han modificado los valores de los siguientes parámetros:

NUM_HORMIGAS = 10
ITER_MAX = 1000

Para las pruebas con los algoritmos GAG y GAGEsp se han modificado los valores de los siguientes parámetros:

GENERACIONES = 500
TAM_POBLACIÓN = 500

Para las pruebas con los algoritmos MAG y MAGEsp se han modificado los valores de los siguientes parámetros:

GENERACIONES = 200
TAM_POBLACIÓN = 30

Para las pruebas con los algoritmos GRASP y GRASPi, se han modificado los valores de los siguientes parámetros:

ALFA = 0.78
NUM_ITERACIONES = 100

Para las pruebas con los algoritmos sim y simi, se mantienen los valores iniciales de sus parámetros.

Justificación de la configuración de los parámetros utilizados

En los algoritmos genéticos y meméticos, hemos realizados pruebas tanto de parámetros como de operadores:

De forma general podemos explicar que la tasa de cruce tiene que ser bastante alta, para producir nuevos individuos, y de esta manera la usamos entre el 0,75 y el 1. Además, la tasa de mutación tiene que ser baja para no modificar de manera grande los individuos creados. Esta se suele encontrar inferior al 0,1. Además, el elitismo ha propiciado una mayor eficacia.

En el problema del TSP con 929 ciudades, el mejor ha sido el algoritmo MAGEsp, con eficacia del 99,86%, tardando un tiempo de 5,989 segundos con el operador de selección Torneo

Determinista, el operador de cruce OX, el operador de mutación Mezcla y el optimizador 2-Opt, usando la versión elitista con un tamaño de población de 10 individuos y 100 generaciones, una tasa de cruce de 0,75 y una tasa de mutación de 0,015. En este problema hemos alcanzado un rango de eficacia de entre el 87% y el 99.86% respecto a ACEi.

En el problema de la Mochila 0-1, hemos alcanzado mayor eficacia, aunque la dimensión es menor. Aun así, en el problema con 200 ítems (difícil), tanto los algoritmos genéticos como meméticos ha alcanzado la *optimalidad*, tardando una media de 1 segundo. Cabe destacar que, para esta instancia del problema, solo hay 12 resultados con eficacia menor al 90% y van desde el 79,34% hasta al 89,87%. Los otros 500 resultados tienen una eficacia superior al 90%, y de estos, unos 400 la tienen superior al 99%. Sin embargo, en el problema con 200 ítems (fácil), se han obtenido peores resultados. Los mejores han sido los algoritmos genéticos, con poblaciones grandes (500) y bastantes generaciones (200 y 500), tardando 1 y 2 segundos respectivamente. Aunque también se han obtenido unos 400 resultados con eficacia superior al 99.95%. Lo único que, en esta instancia del problema, se ha pronunciado más el empeoramiento cuando se ha utilizado el operador de selección ruleta, lo que es debido a la existencia de individuos no factibles creados.

En los algoritmos de colonias de hormigas, nos hemos visto favorecidos en las versiones elitistas con búsqueda local. Esto ha provocado una reducción del tiempo al disminuir los valores de los parámetros de número de hormigas y del tamaño de la población. Además, el valor Beta, suele estar entre 2 y 5.

Según las pruebas realizadas, la modificación de los valores de los parámetros de GRASP, no ha variado de manera significativa los resultados obtenidos.

En cuanto a la metaheurística simulated annealing, se utilizaron los mismos valores iniciales de sus parámetros.

Queremos justificar que hemos utilizado los valores de los parámetros mostrados anteriormente porque algunos daban bastantes buenos resultados. Creemos que, aunque esto se podría considerar injusto porque se hace un número distinto de evaluaciones de la función objetivo con cada algoritmo, no es tan importante, puesto que pensamos que la naturaleza de cada algoritmo juega un papel importante, además del uso de una solución inicial.

Hemos escogido como métricas a valorar, la eficacia y el tiempo de ejecución, puesto que creemos que es lo importante. Aun así, el tiempo con los parámetros que hemos utilizado, puede no ser relevante, y el hecho de que un algoritmo tarde mucho no implica lentitud al obtener buenas soluciones, y para ello, hemos hecho un estudio de la evolución de la eficacia de cada algoritmo en el tiempo, para comprobar que, aunque un algoritmo necesite mucho tiempo para ejecutarse, éste ha podido alcanzar una solución muy buena, incluso mejor que otros algoritmos, en un tiempo mucho menor. Por otro lado, permitimos al algoritmo seguir ejecutándose para que intente alcanzar la *optimalidad*.

7.1. TSP

A continuación, se presentan las comparativas entre los algoritmos utilizados para resolver el TSP. Entre ellos, se puede apreciar, que se incluyen heurísticas específicas para este problema, tal como la del vecino más cercano o el método de inserción.

Para la realización de estas comparativas, se ejecutaron los algoritmos sobre diferentes dimensiones (número de ciudades) del problema. Estas instancias de problemas fueron obtenidas de una página web que ofrece ficheros de prueba. Cada uno de estos contiene la representación de un determinado país. Contienen una cabecera que presenta datos como el nombre o el número de lugares, y a continuación la representación de estos lugares mediante coordenadas, las cuales van precedidas de un índice. Por ejemplo, para Argentina se tiene el siguiente fichero: http://www.math.uwaterloo.ca/tsp/world/ar9152.tsp.
A continuación, se muestran unas cuantas líneas del contenido del fichero ar9152.tsp:

```
NAME : ar9152
COMMENT : 9152 locations in Argentina
COMMENT : Derived from National Imagery and Mapping Agency data
TYPE : TSP
DIMENSION : 9152
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 36266.6667 62550.0000
2 34600.0000 58633.3333
3 51650.0000 72300.0000
4 37800.0000 67683.3333
5 35428.0556 60174.1667
6 34583.3333 68550.0000
7 27383.3333 54766.6667
8 34533.3333 63166.6667
9 23766.6667 64683.3333
10 32383.3333 59666.6667
11 32383.3333 59666.6667
12 25416.6667 65533.3333
13 31383.3333 64183.3333
14 37916.6667 62933.3333
15 37183.3333 60166.6667
16 35600.0000 60900.0000
17 33783.3333 60600.0000
18 35454.1667 60891.3889
19 31650.0000 68400.0000
20 28833.3333 58833.3333
21 26283.3333 64483.3333
22 34642.5000 58618.0556
23 34150.0000 60800.0000
```

Figura 7.1. Fichero que representa un problema con lugares de Argentina

7.1.1. Comparativa de eficacia y tiempo según dimensión del problema

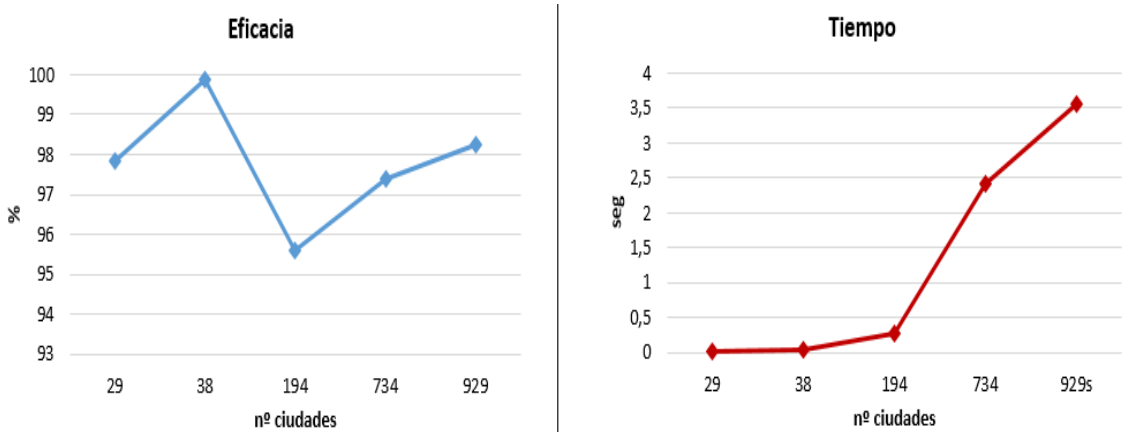
Para realizar este tipo de pruebas hemos ejecutado todos los algoritmos parametrizados en la misma prueba en cada instancia del problema. De manera que, de forma automática podemos saber el coste de cada algoritmo en cada instancia para realizar comparaciones. Por lo que el algoritmo que tenga 100% de eficacia, será el que ha logrado una mejor solución en la instancia, y lo consideraremos como si fuese el óptimo. Así pues, es posible que en diferentes instancias el algoritmo óptimo sea distinto. En cuanto al tiempo, éste es medido según lo que tarde en terminar su ejecución el algoritmo en cuestión.

No hemos utilizado algoritmos de backtracking porque su ejecución sobre problemas a partir de 29 ciudades, tardaría demasiado.

Para obtener los resultados de cada algoritmo, hemos realizado la media con 20 pruebas sobre cada instancia del TSP.

ACEi Opt (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

Dimensión	Eficacia	Tiempo
29 ciudades	97,84%	0,023seg
38 ciudades	99,88%	0,049seg
194 ciudades	95,59%	0,275seg
734 ciudades	97,39%	2,404seg
929 ciudades	98,25%	3,563seg

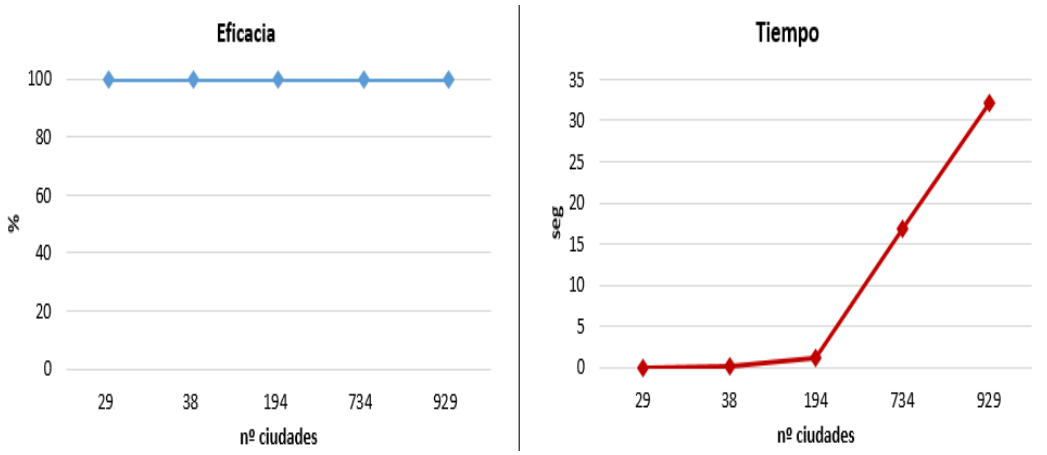


Como era de esperar, debido a que el TSP ha sido modelado sobre grafos, esta versión de algoritmo inspirado en colonia de hormigas, se adapta bien a la resolución de dicho problema. Pero hay que decir que la otra versión de este algoritmo (ACEi OptTot) consigue una mejor eficacia, aunque evidentemente tarda más tiempo debido a su implementación.

Consideremos que esta es la mejor versión del algoritmo, puesto que la diferencia de eficacia no es tan grande como la del tiempo respecto al algoritmo ACEi OptTot. En caso de tener un tiempo ilimitado, lo más probable es que optemos por usar ACEi OptTot, pero creemos que al tardar demasiado respecto a ACEi Opt (sobre todo en problemas grandes), a veces no es recomendable utilizarlo.

ACEi OptTot (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

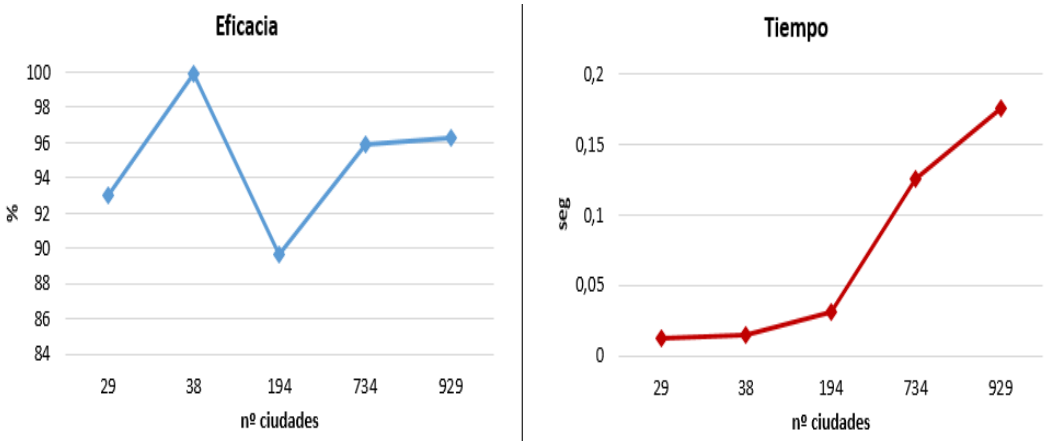
Dimensión	Eficacia	Tiempo
29 ciudades	100%	0,028seg
38 ciudades	100%	0,079seg
194 ciudades	100%	1,189seg
734 ciudades	100%	16,79seg
929 ciudades	100%	32,229seg



Como ya dijimos en el análisis para el algoritmo anterior, la diferencia de eficacia entre ese algoritmo y este no es tan grande como la del tiempo.
Debemos apuntar que, a pesar de la penalización en tiempo, con esta versión alcanzamos la mejor eficacia respecto a los otros algoritmos.

GAG (Algoritmo genético generacional)

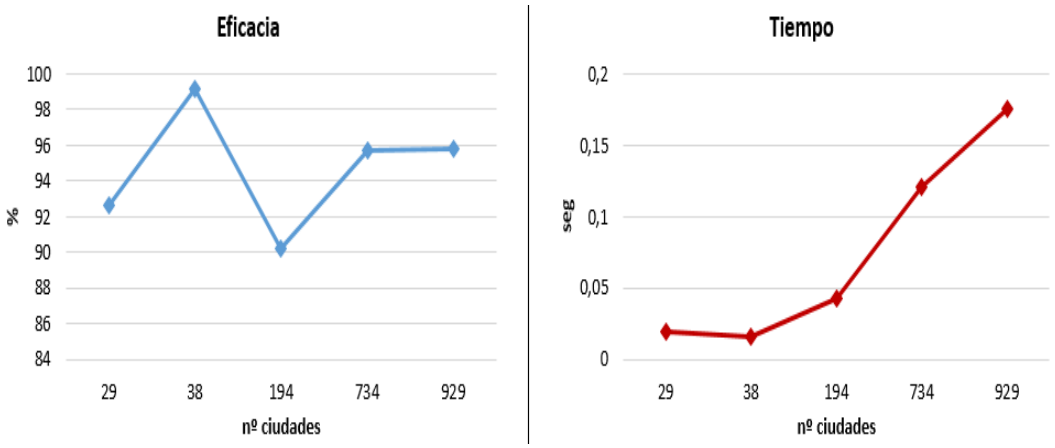
Dimensión	Eficacia	Tiempo
29 ciudades	92,96%	0,013seg
38 ciudades	99,93%	0,015seg
194 ciudades	89,64%	0,031seg
734 ciudades	95,88%	0,125seg
929 ciudades	96,28%	0,175seg



Es un algoritmo bastante rápido, y alcanza una alta eficacia, aunque no alcance el óptimo.

GAGESP (Algoritmo genético generacional específico)

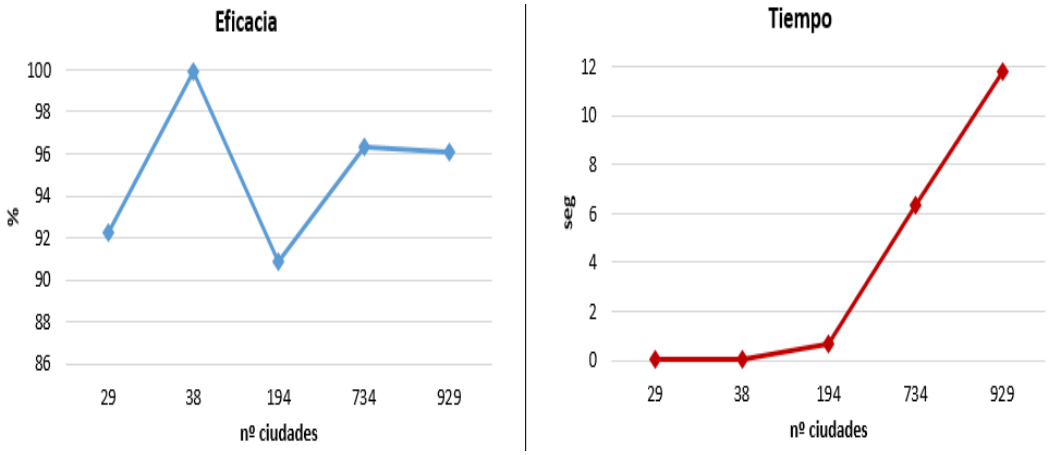
Dimensión	Eficacia	Tiempo
29 ciudades	92,68%	0,019seg
38 ciudades	99,12%	0,016seg
194 ciudades	90,19%	0,043seg
734 ciudades	95,75%	0,121seg
929 ciudades	95,8%	0,175seg



Con esta versión de algoritmo genético generacional, hemos conseguido mejorar un poco la eficacia del algoritmo GAG anterior. No sabemos a qué puede ser debido, porque la única diferencia respecto al anterior es la forma de instanciar el algoritmo, pero la implementación en sí, es la misma. La razón de ponerlo, es porque nos ha parecido curioso este hecho.

MAG (Algoritmo memético generacional)

Dimensión	Eficacia	Tiempo
29 ciudades	92,22%	0,047seg
38 ciudades	99,93%	0,057seg
194 ciudades	90,9%	0,652seg
734 ciudades	96,35%	6,312seg
929 ciudades	96,12%	11,774seg

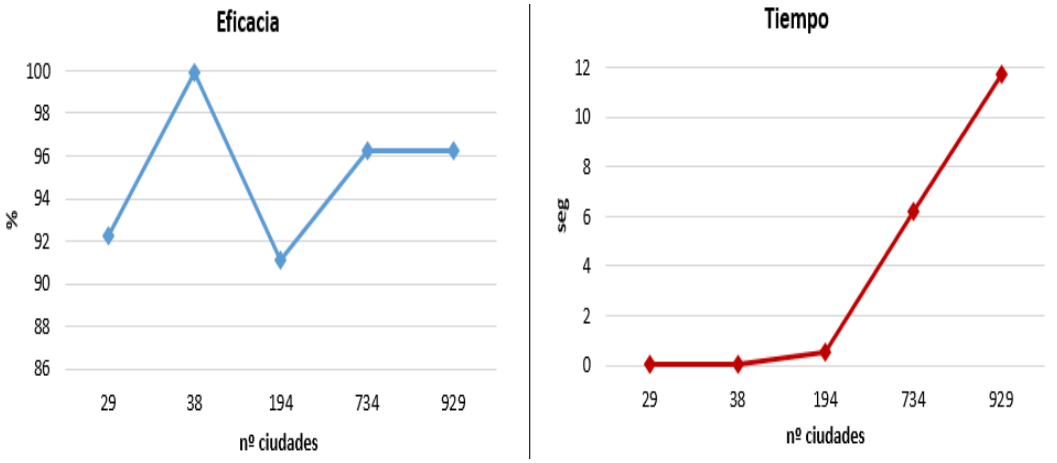


Con el fin de sacar partido a la eficacia de los algoritmos evolutivos, utilizamos la vertiente memética de los algoritmos genéticos, la cual incluye una búsqueda local para mejorar todos los individuos (soluciones) conseguidos.
Con el factor de *memetismo*, la eficacia es algo más alta respecto a la versión sin *memetismo* (GAG), aunque con un tiempo bastante mayor, en problemas de gran dimensión.

Dentro de la parametrización de la metaheurística, podemos hacer que esta sea elitista o no, lo que provoca un pequeño aumento en su eficacia. Para este experimento hemos utilizado una versión elitista.

MAGESP (Algoritmo memético generacional específico)

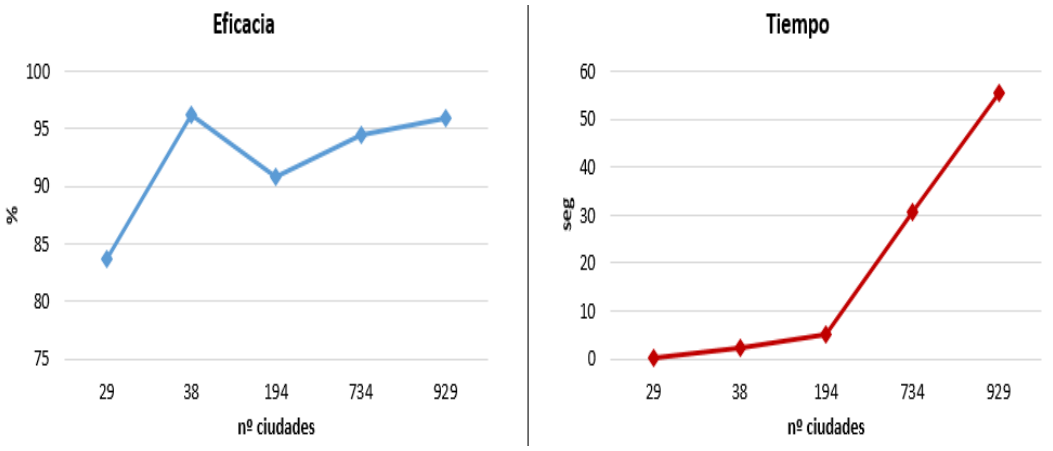
Dimensión	Eficacia	Tiempo
29 ciudades	92,22%	0,029seg
38 ciudades	99,93%	0,058seg
194 ciudades	91,08%	0,56seg
734 ciudades	96,28	6,229seg
929 ciudades	96,22%	11,746seg



Este caso es similar a lo que ocurre con el algoritmo GAG respecto al algoritmo GAGEsp.

GRASPi (Procedimiento aleatorio voraz de búsqueda adaptativa, con solución inicial de partida)

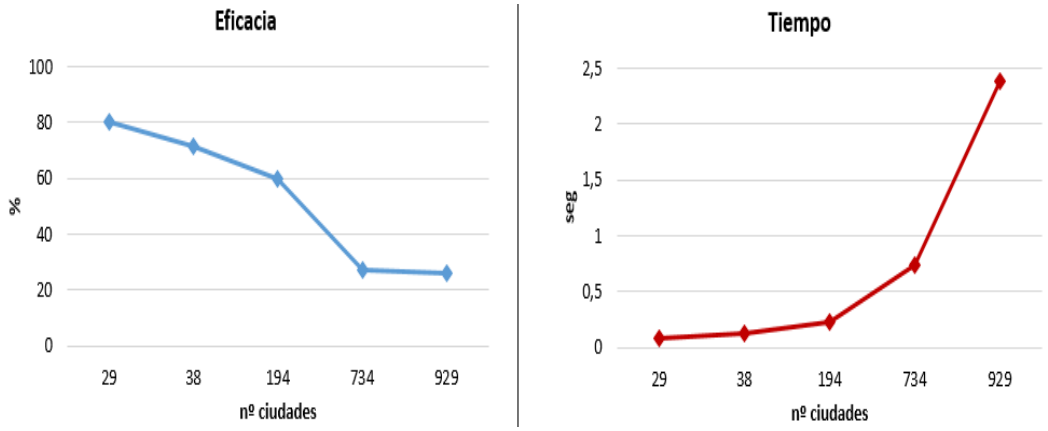
Dimensión	Eficacia	Tiempo
29 ciudades	83,68%	0,414seg
38 ciudades	96,18%	2,276seg
194 ciudades	90,89%	5,002seg
734 ciudades	94,46%	30,69seg
929 ciudades	95,84%	55,479seg



Es una metaheurística lenta para tamaños grandes del problema, compensando dicha lentitud por su eficacia. Es una buena metaheurística, muy fácil de implementar que ofrece buenos resultados, aunque necesita a veces de un gran número de iteraciones para ello, dado su carácter aleatorio.

sim (Algoritmo de enfriamiento simulado)

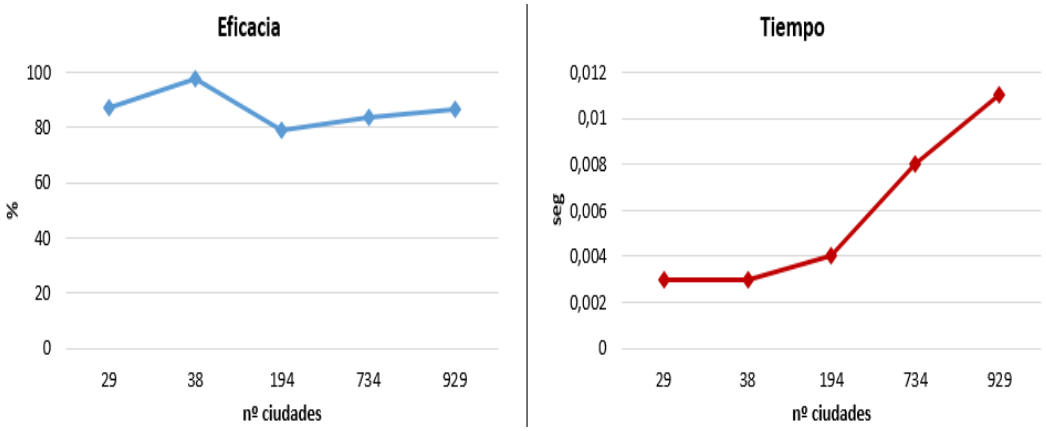
Dimensión	Eficacia	Tiempo
29 ciudades	80,42%	0,086seg
38 ciudades	71,32%	0,128seg
194 ciudades	59,52%	0,229seg
734 ciudades	27,23%	0,741seg
929 ciudades	25,9%	2,379seg



Según aumenta la dimensión del problema la eficacia baja drásticamente llegando a mínimos de ineficacia.

simi (Algoritmo de enfriamiento simulado, con solución inicial de partida)

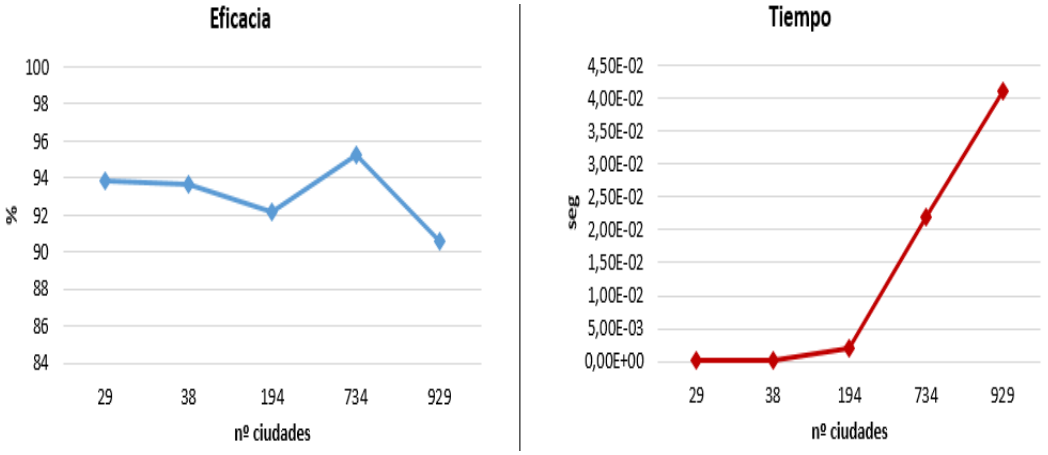
Dimensión	Eficacia	Tiempo
29 ciudades	87,1%	0,003seg
38 ciudades	97,65%	0,003seg
194 ciudades	79,05%	0,004seg
734 ciudades	83,59%	0,008seg
929 ciudades	86,53%	0,011seg



Con una solución inicial obtenida mediante la heurística del vecino más próximo, ha aumentado de manera considerable su eficacia en problemas grandes, pasando de valores mínimos anteriores que rondaban el 30% de eficacia a alrededor del 90% y disminuyendo su tiempo. En algunas pruebas se pudo observar que su tiempo de ejecución era demasiado rápido, independientemente del número de ciudades del problema, lo que nos hace pensar que el algoritmo en esos casos podría haberse quedado anclado en algún óptimo local demasiado rápido (convergiendo).

INSAP Opt (Heurística optimizada de inserción posterior implementada con arrays)

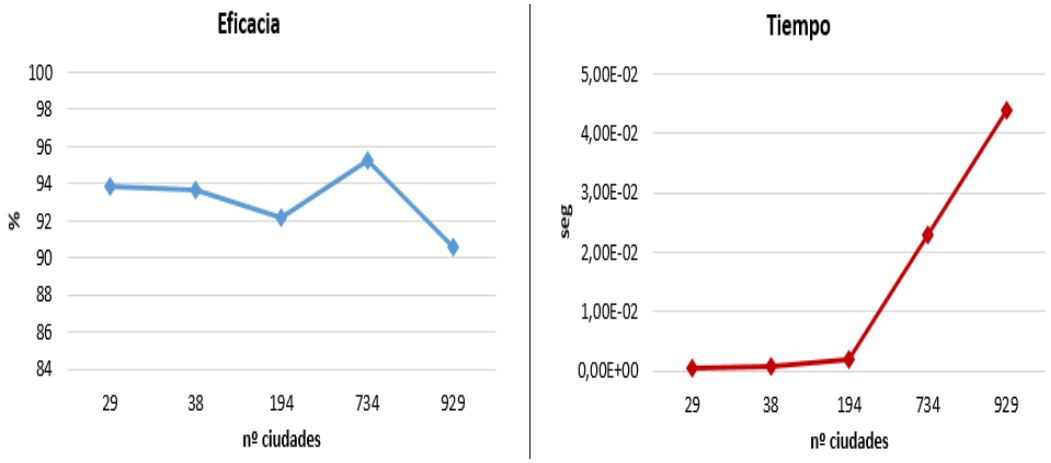
Dimensión	Eficacia	Tiempo
29 ciudades	93,88%	0,00014seg
38 ciudades	93,7%	0,00019seg
194 ciudades	92,19%	0,002seg
734 ciudades	95,21%	0,022seg
929 ciudades	90,59%	0,041seg



Se puede apreciar que esta heurística tiene un desempeño muy bueno, tanto en eficacia como en tiempo para las instancias utilizadas del problema. Debido a esto, optamos por no utilizarla para obtener soluciones iniciales para las metaheurísticas, si no por utilizarla como un método más para la resolución del TSP.

INSCP Opt (Heurística optimizada de inserción posterior implementada con conjuntos)

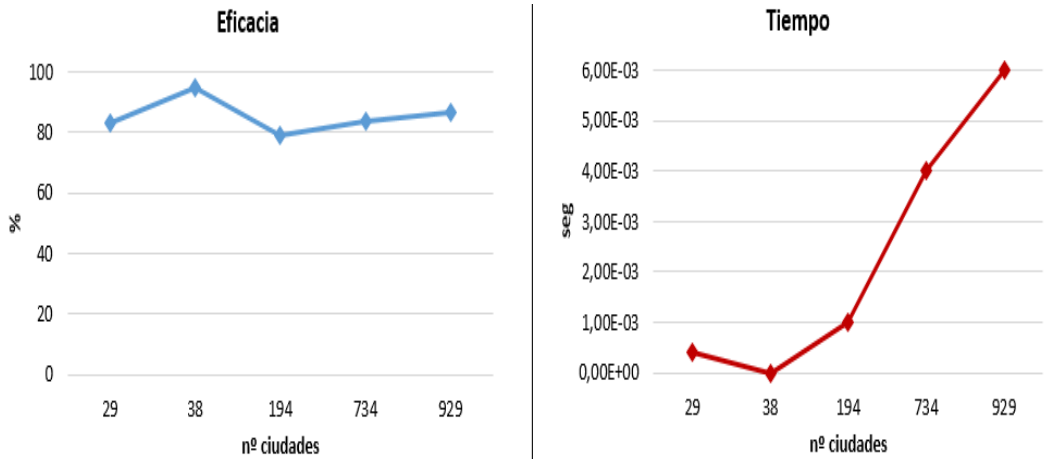
Dimensión	Eficacia	Tiempo
29 ciudades	93,88%	0,0004seg
38 ciudades	93,7%	0,0009seg
194 ciudades	92,19%	0,002seg
734 ciudades	95,21%	0,023seg
929 ciudades	90,59%	0,044seg



Similar a INSAP Opt. La eficacia obtenida por ambos algoritmos en cada instancia del problema, es la misma, cambiando mínimamente los tiempos. Creemos que esto se debe a la implementación, puesto que aquí utilizamos conjuntos, mientras que en INSAP Opt, utilizamos arrays.

Vecino (Vecino más cercano)

Dimensión	Eficacia	Tiempo
29 ciudades	83,27%	0,0004seg
38 ciudades	94,63%	0seg
194 ciudades	78,8%	0,001seg
734 ciudades	83,76%	0,004seg
929 ciudades	86,66%	0,006seg



Hemos utilizado esta heurística para obtener una solución inicial para guiar a ciertos algoritmos. Como se puede ver, su eficacia es relativamente alta, aunque no siempre, y su tiempo es pequeño.

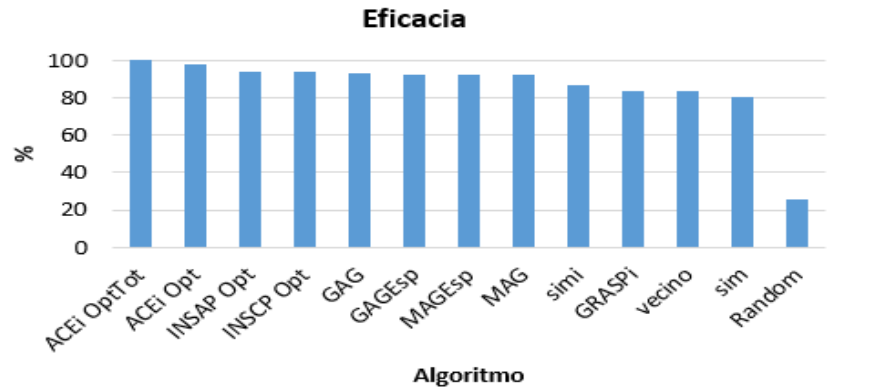
7.1.2. Comparativa conjunta según dimensión del problema

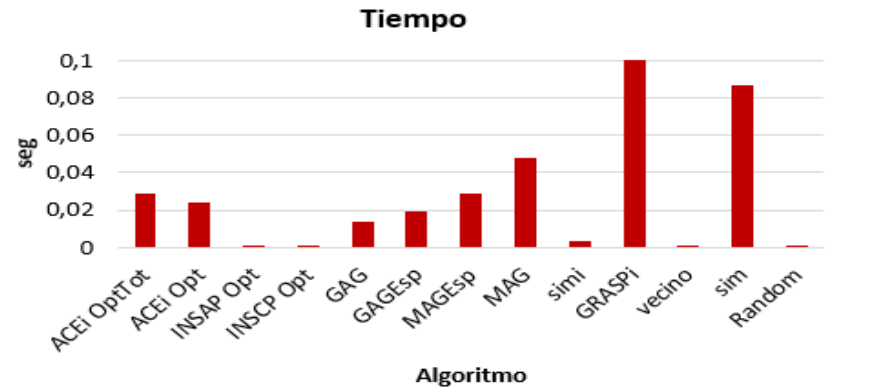
En este tipo de comparativa no existe límite de tiempo para la ejecución de los algoritmos, por lo que, se utilizan como valores de comparación la eficacia de la solución obtenida y el tiempo empleado para ello de cada algoritmo y se comparan con los correspondientes valores obtenidos por el mejor algoritmo.

Al igual que en la sección anterior, no hemos utilizado algoritmos de backtracking porque su ejecución sobre problemas a partir de 29 ciudades tardaría demasiado.

29 ciudades

Algoritmo	Eficacia %	Coste	T (seg)
ACEi OptTot	100	27643,18	0,02885714
ACEi Opt	97,84	28253,9	0,02380952
INSAP Opt	93,88	29444,63	1,43E-04
INSCP Opt	93,88	29444,63	4,29E-04
GAG	92,96	29735,89	0,0137619
GAGEsp	92,68	29827,34	0,01957143
MAGEsp	92,22	29975,08	0,02914286
MAG	92,22	29975,08	0,04790476
simi	87,1	31738,17	0,00361905
GRASPi	83,68	33036,22	0,41490476
vecino	83,27	33197,9	4,29E-04
sim	80,42	34372,89	0,08666667
Random	25,37	108965,33	9,52E-05



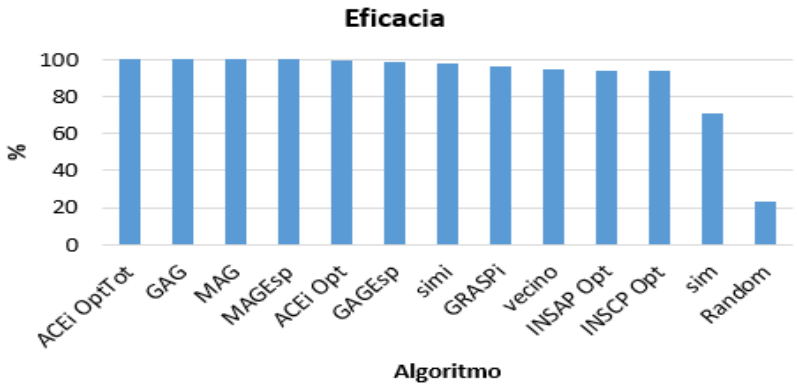


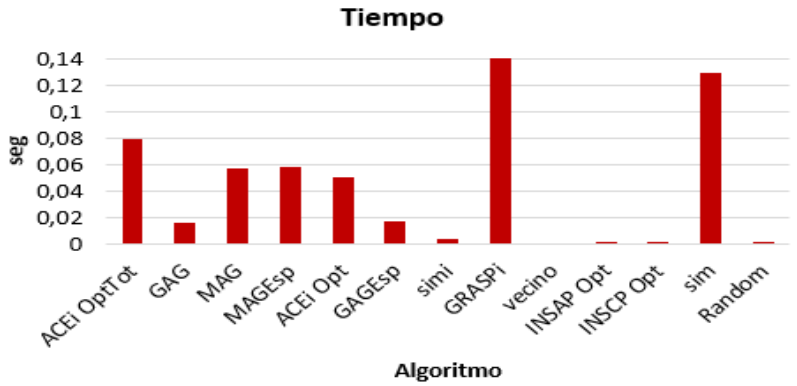
Para esta instancia del problema, los algoritmos ACEi Opt y ACEI OptTot han sido los mejores en cuanto a eficacia, además de haber conseguido buenos tiempos. Por otro lado, la relación eficacia/tiempo del algoritmo GRASPi no ha sido la mejor, sobre todo por el tiempo

utilizado, siendo este, el mayor de entre los obtenidos por los demás algoritmos metaheurísticos. De entre los otros algoritmos, el que más destaca es el de inserción, que ha conseguido resultados muy buenos en unos tiempos también muy buenos.

38 ciudades

Algoritmo	Eficacia %	Coste	T (seg)
ACEi OptTot	100	6659,43	0,07980952
GAG	99,93	6663,91	0,01590476
MAG	99,93	6664,11	0,05742857
MAGEsp	99,93	6664,11	0,05828571
ACEi Opt	99,88	6667,14	0,04990476
GAGEsp	99,12	6718,8	0,01680952
simi	97,65	6819,94	0,00385714
GRASPi	96,18	6923,68	2,27604762
vecino	94,63	7037,23	0
INSAP Opt	93,7	7107,27	1,90E-04
INSCP Opt	93,7	7107,27	9,52E-04
sim	71,32	9337,8	0,12890476
Random	23,52	28319,23	1,90E-04

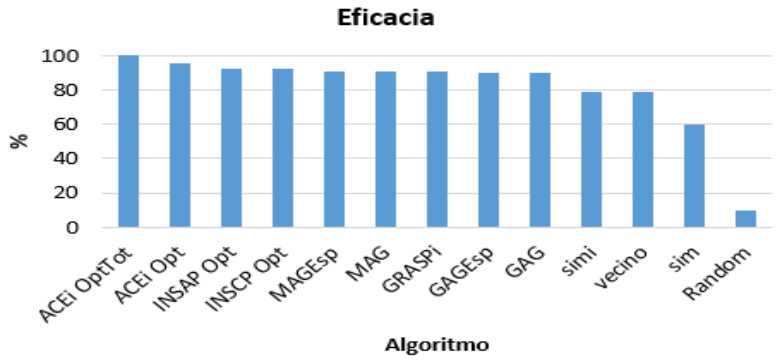


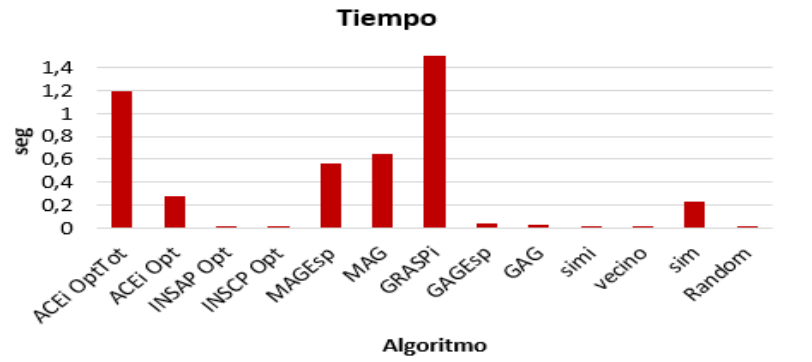


En este caso, el algoritmo ACEi OptTot ha sido de nuevo el mejor, seguido del algoritmo GAG, que es el que más destaca de entre los algoritmos genéticos. Pero es justo decir que, en términos generales todos los algoritmos han alcanzado una buena eficacia, en un tiempo razonable, excepto el algoritmo Random, debido a su propia naturaleza. Por otro lado, el algoritmo sim ha quedado bastante lejos del resto.

194 ciudades

Algoritmo	Eficacia %	Coste	T (seg)
ACEi OptTot	100	9500,2	1,18938095
ACEi Opt	95,59	9938,1	0,27504762
INSAP Opt	92,19	10305,36	0,00228571
INSCP Opt	92,19	10305,36	0,00285714
MAGEsp	91,08	10430,26	0,56066667
MAG	90,9	10451,83	0,65204762
GRASPi	90,89	10452,63	5,00271429
GAGEsp	90,19	10533,46	0,0432381
GAG	89,64	10598,67	0,0312381
simi	79,05	12018,31	0,00495238
vecino	78,8	12055,67	0,00133333
sim	59,52	15960,67	0,22971429
Random	10,19	93191,15	1,90E-04

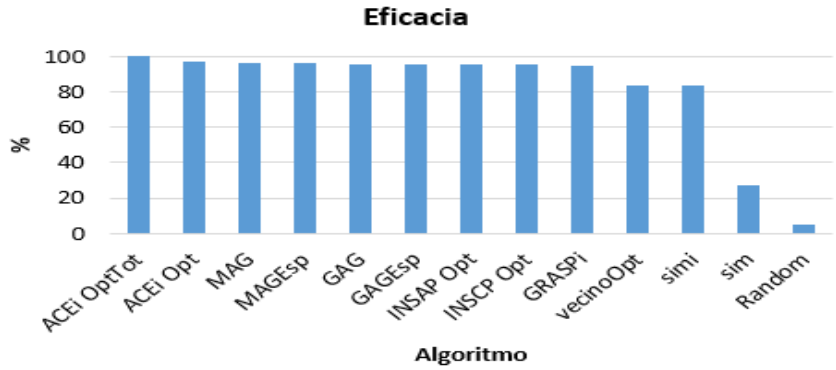


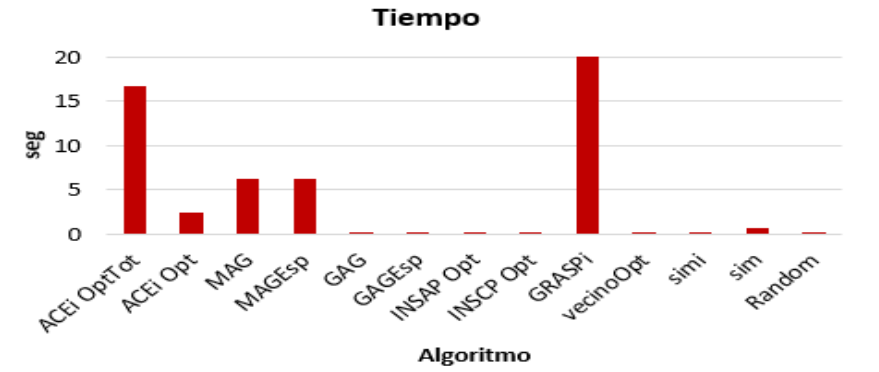


Al aumentar la dimensión del problema, podemos apreciar, más diferencias entre los algoritmos. Los dos mejores algoritmos en cuanto a eficacia son ACEi OptTot y ACEi Opt. El resto de metaheurísticas, poseen una eficacia parecida, menos sim y simi.

734 ciudades

Algoritmo	Eficacia %	Coste	T (seg)
ACEi OptTot	100	83389,05	16,7909048
ACEi Opt	97,39	85627,84	2,40419048
MAG	96,35	86551,18	6,31285714
MAGEsp	96,28	86611,64	6,22928571
GAG	95,88	86975,4	0,125
GAGEsp	95,75	87089	0,12171429
INSAP Opt	95,21	87581,79	0,02247619
INSCP Opt	95,21	87581,79	0,02361905
GRASPi	94,46	88280,51	30,6904762
vecinoOpt	83,76	99560,31	0,00495238
simi	83,59	99761,18	0,00895238
sim	27,23	306205,88	0,74180952
Random	5,1	1633681,72	1,90E-04



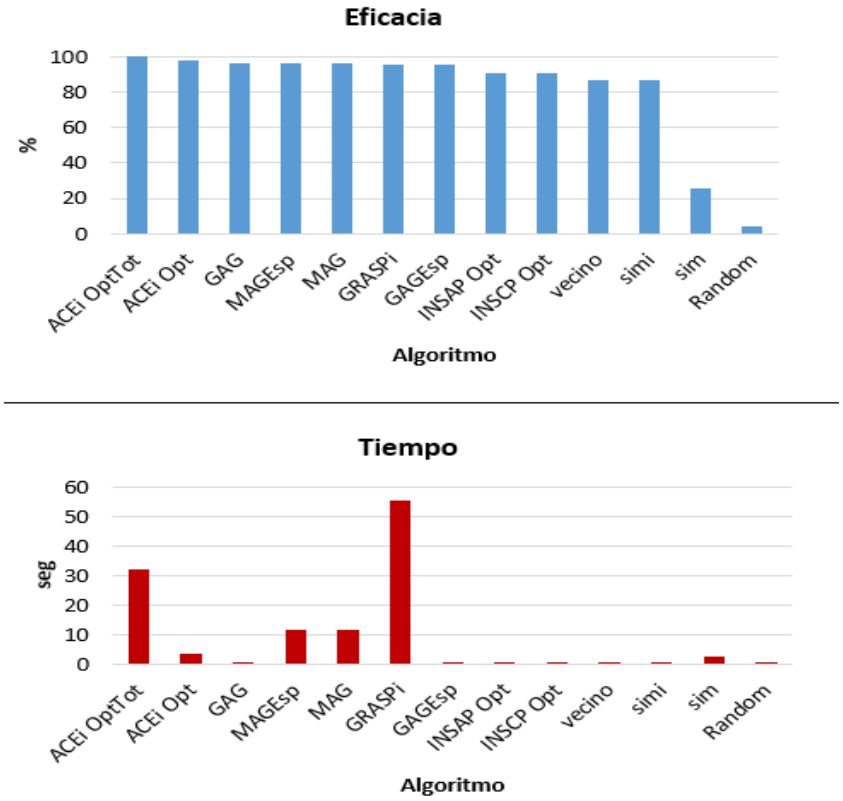


En esta instancia, ACEi OptTot sigue siendo mejor el algoritmo en cuanto a eficacia, sin embargo, su tiempo es el segundo peor, solamente superado por GRASPi, el cual obtiene una buena eficacia.

Los algoritmos genéticos han aumentado su eficacia respecto a otros algoritmos y han subido puestos en la clasificación de los algoritmos. Esto parece indicar que, al aumentar la dimensión del problema, siguen siendo buenos.

929 ciudades

Algoritmo	Eficacia %	Coste	T (seg)
ACEi OptTot	100	100553,11	32,229381
ACEi Opt	98,25	102345,93	3,56319048
GAG	96,28	104436,8	0,1752381
MAGEsp	96,22	104498,77	11,746
MAG	96,12	104613,12	11,7741429
GRASPi	95,84	104917,26	55,479619
GAGEsp	95,8	104958,6	0,17504762
INSAP Opt	90,59	110993,71	0,04157143
INSCP Opt	90,59	110993,71	0,0442381
vecino	86,66	116025,66	0,00685714
simi	86,53	116199,62	0,01180952
sim	25,9	388225,24	2,37909524
Random	4,22	2381233,6	5,71E-04



En esta última instancia, se han conseguido resultados similares a la anterior. Cabe resaltar que, aunque algoritmo simi utiliza como solución de partida una obtenida mediante el algoritmo del vecino más cercano, es ligeramente peor que este. Esto nos lleva a pensar que probablemente simi no ha sido capaz de mejorar la solución inicial del algoritmo del vecino más cercano.

Conclusiones

Se puede observar que las dos mejores metaheurísticas son las inspiradas en colonias de hormigas (ACEi Opt y ACEi OptTot). Esto puede ser debido a que son metaheurísticas orientadas a grafos.

Después de estos dos, los algoritmos genéticos (GAG, GAGEsp, MAG y MAGEsp) son los que mejores resultados consiguen.
Al aumentar de manera drástica el tiempo con los meméticos (en ocasiones hasta un poco más del doble), la eficacia consigue un pequeño aumento.

En cuanto a la metaheurística GRASPi, esta obtiene buenos resultados, pero en contra, su tiempo de ejecución aumenta bastante cada vez que aumenta la dimensión del problema.

La metaheurística sim, al no usar una solución inicial, no obtiene tan buenos resultados en comparación con la metaheurística simi, la cual utiliza una solución inicial obtenida mediante la

heurística del vecino más próximo. Además, también se nota la diferencia en cuanto al tiempo de ejecución de ambos algoritmos según la dimensión del problema, siendo el tiempo de ejecución de sim siempre mayor que el de simi.

7.1.3. Comparativas en un segundo

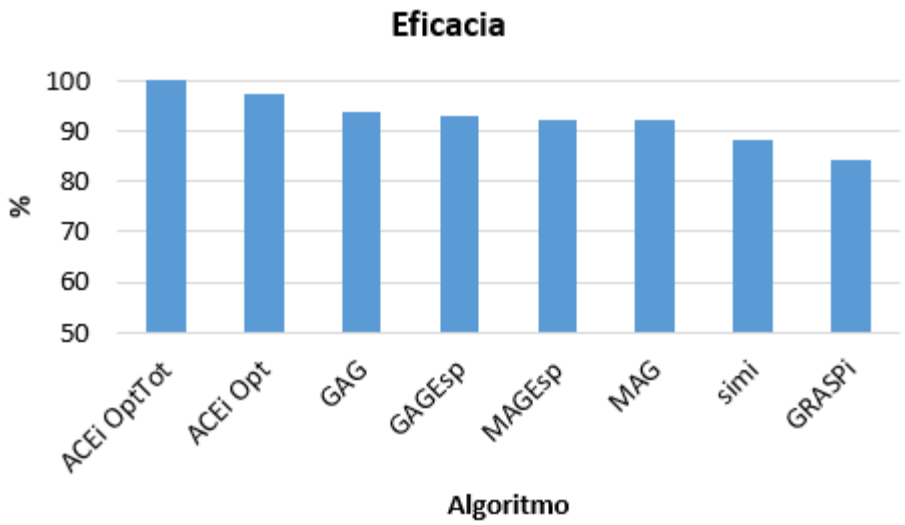
Los algoritmos que más tardaron en la sección de comparativas anterior, fueron GRASPi y ACEi OptTot, los cuales necesitan incluso de minutos para lograr su mejor resultado, lo que nos hace preguntarnos si son tan buenos.

Para comprobarlo, a continuación, mostramos la eficacia de todos los algoritmos para diferentes dimensiones del problema al cabo de un segundo de ejecución. Para comparar dicha eficacia, establecemos como cota superior, una solución alcanzada mediante el algoritmo ACEi Opt después de un número de iteraciones determinado. Por lo que, aunque la eficacia de la solución obtenida por algunos algoritmos llegue al 100%, quizá no sea dicha solución la óptima del problema.

Para realizar estas pruebas, hemos tenido que modificar los criterios de parada de los algoritmos, y usar un reloj para establecer cuando ha transcurrido un segundo. Cabe destacar que este segundo es consumido únicamente en la parte de ejecución, no contabilizamos el tiempo usado para inicializar el algoritmo. También hay que apuntar que los resultados obtenidos son la media de 20 pruebas realizadas con cada algoritmo.

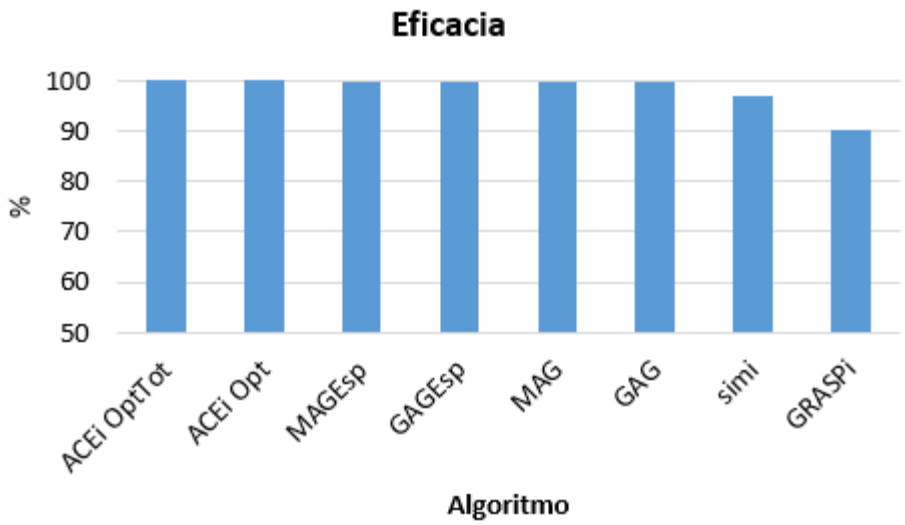
29 ciudades

Algoritmo	Eficacia	Coste
ACEi OptTot	100%	27601,17
ACEi Opt	97,49%	28311,63
GAG	93,96%	29376,02
GAGEsp	93,03%	29670,53
MAGEsp	92,08%	29975,08
MAG	92,08%	29975,08
simi	88,27%	31269,06
GRASPi	84,15%	32801,12



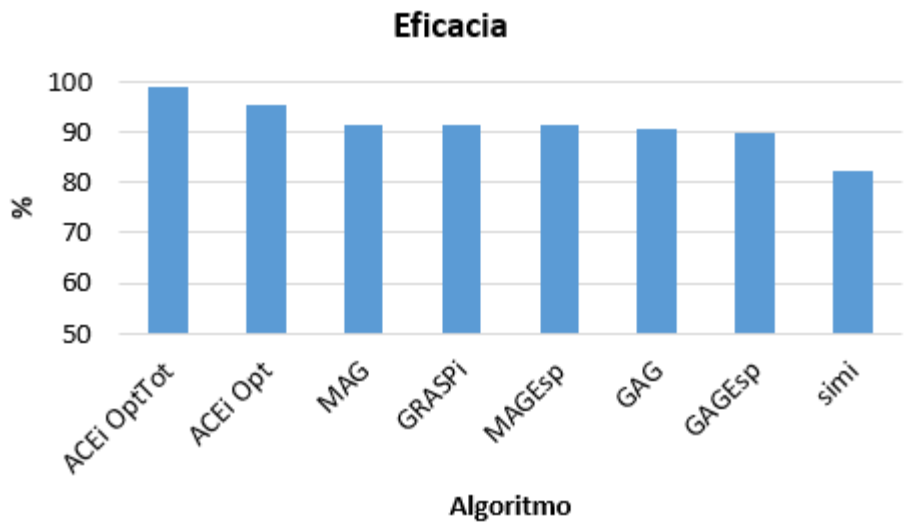
38 ciudades

Algoritmo	Eficacia	Coste
ACEi OptTot	100%	6659,43
ACEi Opt	99,99%	6660,33
MAGEsp	99,94%	6663,73
GAGEsp	99,93%	6663,86
MAG	99,93%	6664,11
GAG	99,93%	6664,25
simi	97,01%	6864,68
GRASPi	90,17%	7385,7



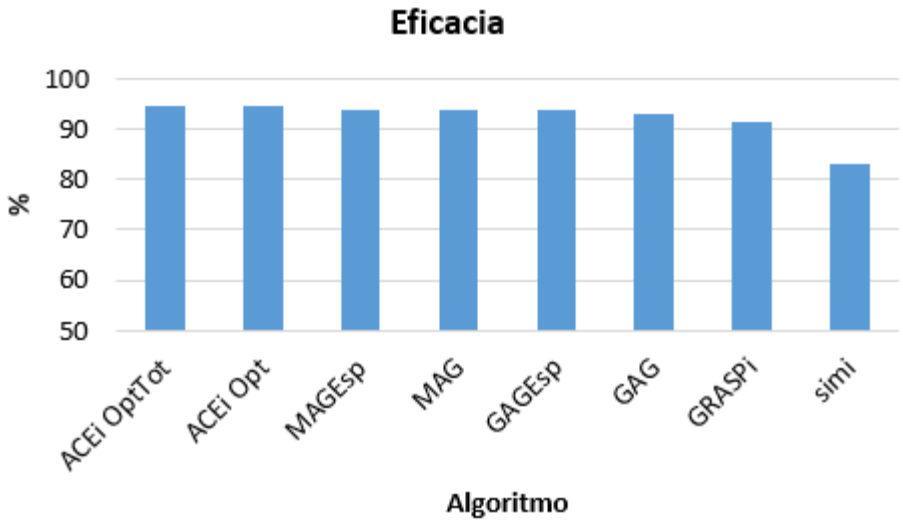
194 ciudades

Algoritmo	Eficacia	Coste
ACEi OptTot	98,79%	9651,14
ACEi Opt	95,59%	9973,76
MAG	91,48%	10422,01
GRASPi	91,24%	10449,32
MAGEsp	91,24%	10449,96
GAG	90,57%	10526,54
GAGEsp	89,83%	10613,39
simi	82,46%	11562,85



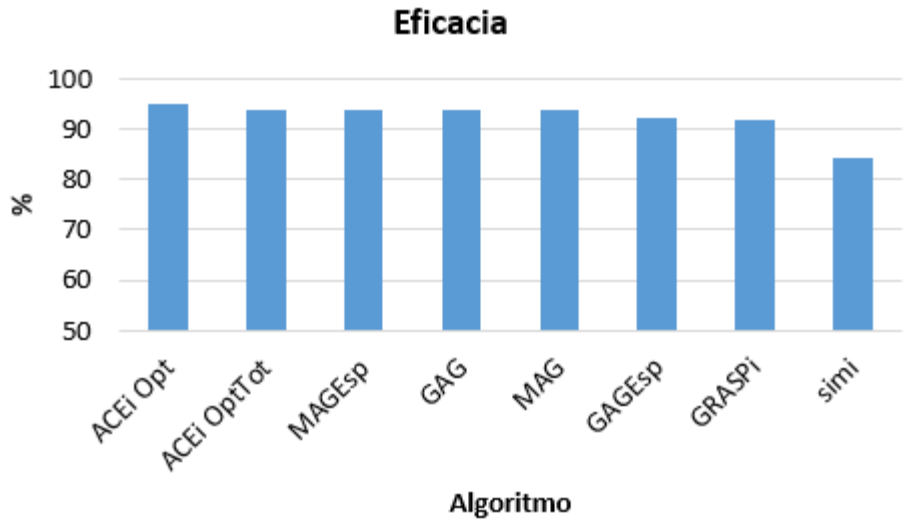
734 ciudades

Algoritmo	Eficacia	Coste
ACEi OptTot	94,72%	85973,65
ACEi Opt	94,41%	86257,16
MAGEsp	93,99%	86639,86
MAG	93,95%	86674,86
GAGEsp	93,93%	86692,09
GAG	93,21%	87367,81
GRASPi	91,28%	89210,08
simi	82,99%	98125,88



929 ciudades

Algoritmo	Eficacia	Coste
ACEi Opt	95,12%	102934,02
ACEi OptTot	93,96%	104195,48
MAGEsp	93,76%	104424,36
GAG	93,72%	104467,82
MAG	93,72%	104471,43
GAGEsp	92,34%	106026,42
GRASPi	91,85%	106592,55
simi	84,4%	116006,95



7.1.4. Progresión de la evolución de la eficacia (individual)

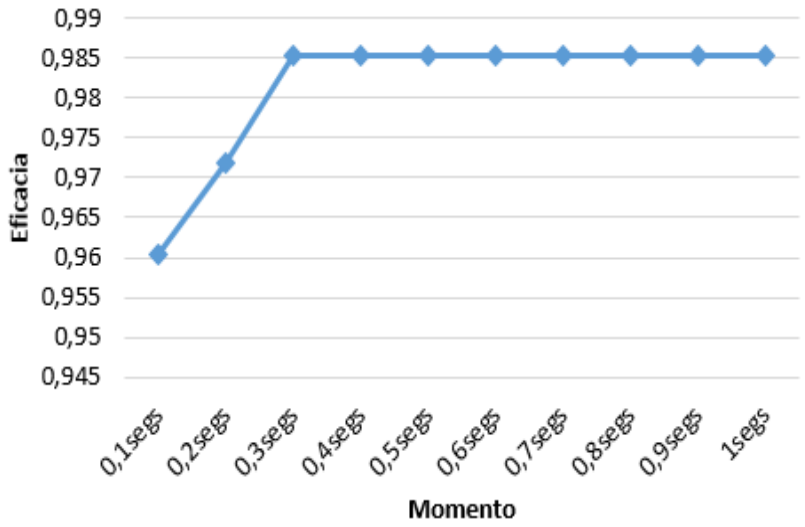
A continuación, vamos a estudiar cómo crece la eficacia de cada algoritmo durante un segundo. Para ello hemos sacado la solución obtenida por cada algoritmo en cada décima de segundo, para ver qué eficacia logran. Este tiempo lo tenemos que considerar únicamente como ejecución, pues no incluye la inicialización. Esto es importante puesto que se pueden dar casos, en los que la preparación o inicialización de determinados algoritmos con unos parámetros muy altos, dure incluso más de un segundo. Para ello, hemos decrementado los valores de estos, para reducir el coste de inicialización, por ejemplo, a 2 décimas de segundo.

Para realizar este experimento, únicamente hemos utilizado una instancia del problema con 929 ciudades, pues deseábamos comprobar el comportamiento de los algoritmos sobre un problema grande.

929 ciudades

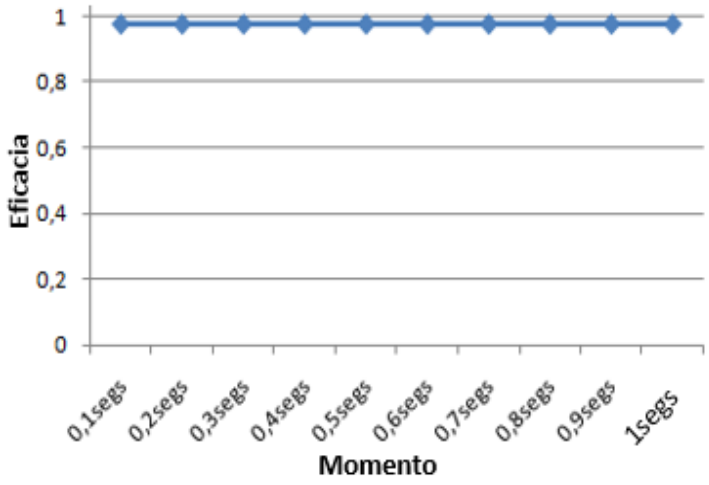
ACEi OptTot (Metaheurística optimizada con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

Momento	Eficacia
0,1segs	0,96049775
0,2segs	0,9719097
0,3segs	0,98523957
0,4segs	0,98523957
0,5segs	0,98523957
0,6segs	0,98523957
0,7segs	0,98523957
0,8segs	0,98523957
0,9segs	0,98523957
1segs	0,98523957



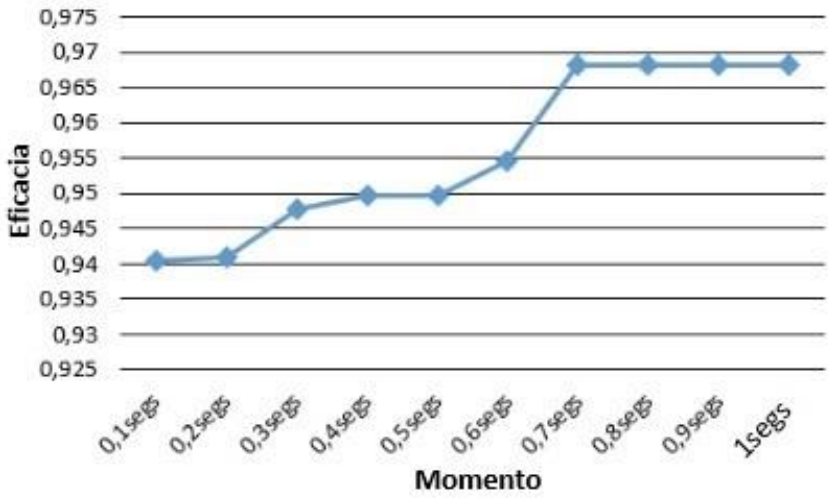
GAG (Algoritmo genético generacional)

Momento	Eficacia
0,1segs	0,97551244
0,2segs	0,97551244
0,3segs	0,97551244
0,4segs	0,97551244
0,5segs	0,97551244
0,6segs	0,97551244
0,7segs	0,97551244
0,8segs	0,97551244
0,9segs	0,97551244
1segs	0,97551244



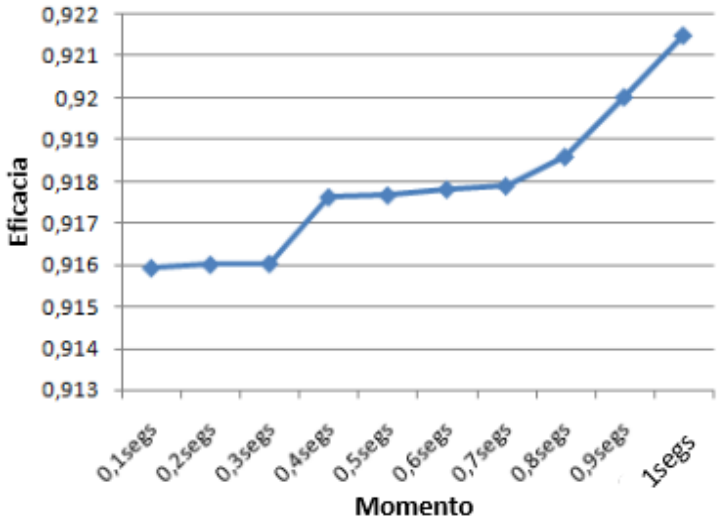
GRASPi (Procedimiento aleatorio voraz de búsqueda adaptativa, con solución inicial de partida)

Momento	Eficacia
0,1segs	0,94042579
0,2segs	0,94083179
0,3segs	0,94771176
0,4segs	0,94972793
0,5segs	0,94972793
0,6segs	0,95452769
0,7segs	0,96831731
0,8segs	0,96831731
0,9segs	0,96831731
1segs	0,96831731



Simi (Algoritmo de enfriamiento simulado, con solución inicial de partida)

Momento	Eficacia
0,1segs	0,91591582
0,2segs	0,91599769
0,3segs	0,91601959
0,4segs	0,91761831
0,5segs	0,9176622
0,6segs	0,91780497
0,7segs	0,91788494
0,8segs	0,91858907
0,9segs	0,9200228
1segs	0,92150691



Conclusiones

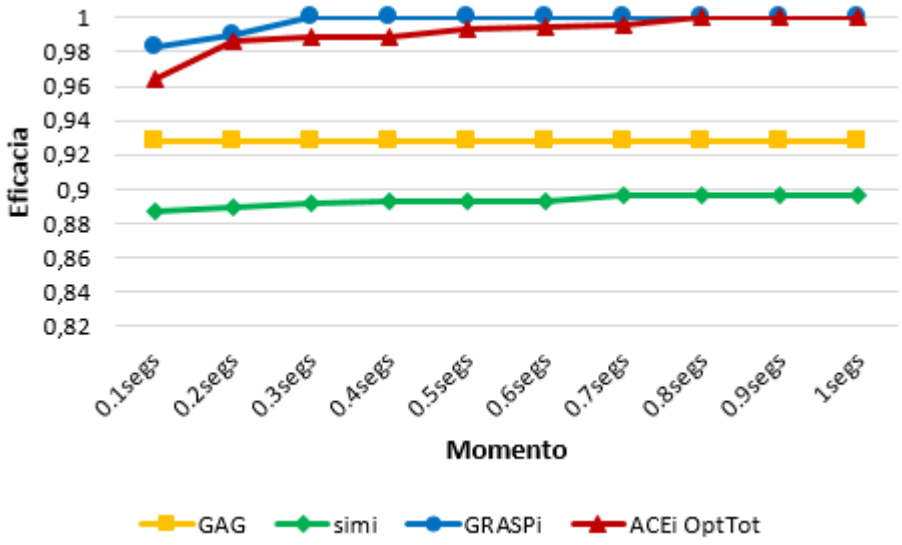
Como vemos con estas gráficas, el crecimiento de la eficacia es gradual excepto en el algoritmo GAG. Esto puede ser debido a que es un algoritmo lento y que, según el tamaño de la población inicial, una iteración puede tardar mucho, y quedarse en ella.

7.1.5. Progresión de la evolución de la eficacia (todos los algoritmos sobre una misma gráfica)

En este tipo de comparativas, vamos a ver el crecimiento de la eficacia de los algoritmos durante un segundo sobre una misma gráfica para distintos tamaños del problema. Para ello creímos razonable utilizar tres problemas de dimensión grande.

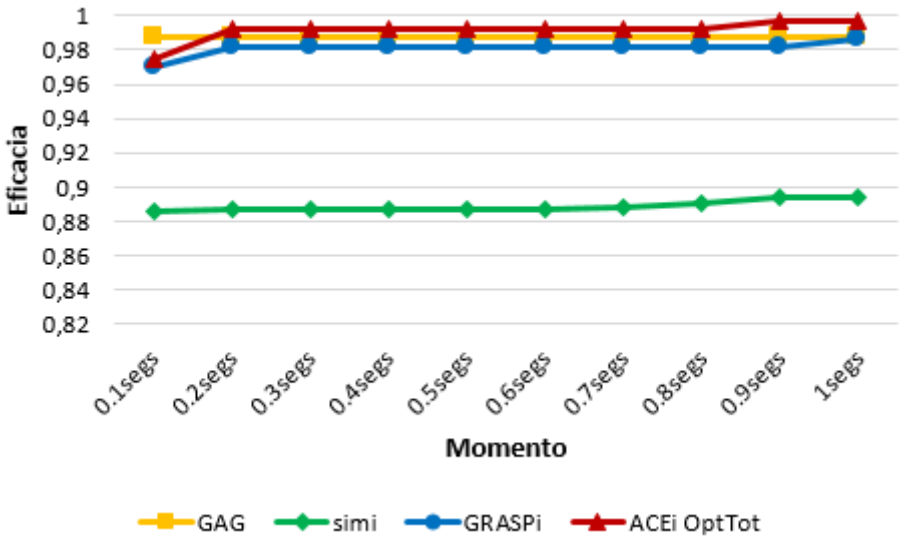
194 ciudades

Momento	GAG	simi	GRASPi	ACEi OptTot
0.1segs	0,92758382	0,88671685	0,98251804	0,96439444
0.2segs	0,92758382	0,88942047	0,99011654	0,98685097
0.3segs	0,92758382	0,89116865	1	0,98847646
0.4segs	0,92758382	0,89232443	1	0,98847646
0.5segs	0,92758382	0,89232443	1	0,99349006
0.6segs	0,92758382	0,89232443	1	0,99493015
0.7segs	0,92758382	0,8962809	1	0,9953312
0.8segs	0,92758382	0,8962809	1	1
0.9segs	0,92758382	0,89683451	1	1
1segs	0,92758382	0,89683451	1	1



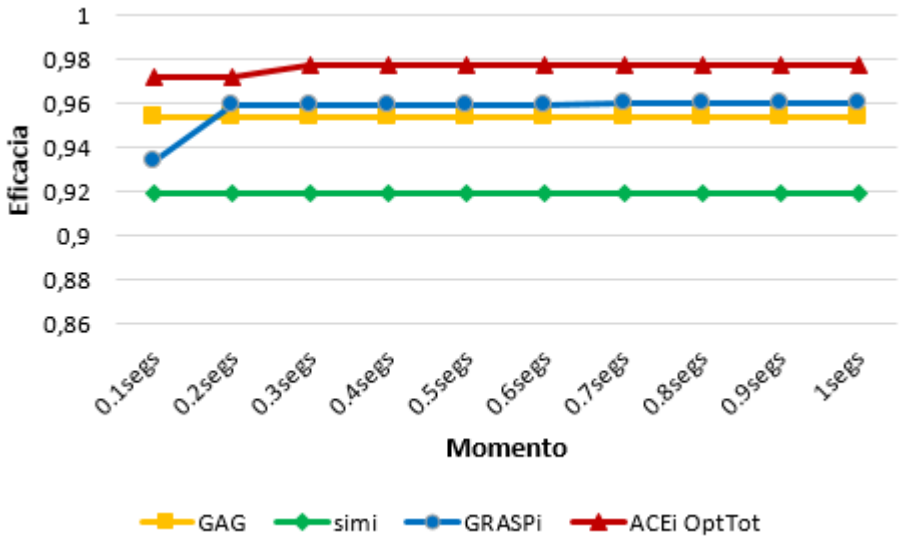
734 ciudades

Momento	GAG	simi	GRASPi	ACEi OptTot
0.1segs	0,98808135	0,88604467	0,96990217	0,9743638
0.2segs	0,98808135	0,88678844	0,9811277	0,99272326
0.3segs	0,98808135	0,88678844	0,9811277	0,99272326
0.4segs	0,98808135	0,88678844	0,9811277	0,99272326
0.5segs	0,98808135	0,88678844	0,9811277	0,99272326
0.6segs	0,98808135	0,88699359	0,9811277	0,99272326
0.7segs	0,98808135	0,88872482	0,9811277	0,99272326
0.8segs	0,98808135	0,89067744	0,9811277	0,99272326
0.9segs	0,98808135	0,89409944	0,9811277	0,99738397
1segs	0,98808135	0,89409944	0,9861064	0,99738397



929 ciudades

Momento	GAG	simi	GRASPi	ACEi OptTot
0.1segs	0,95368951	0,91939349	0,93398903	0,972022
0.2segs	0,95368951	0,91939349	0,9591658	0,972022
0.3segs	0,95368951	0,91939349	0,9591658	0,97753957
0.4segs	0,95368951	0,91939349	0,9591658	0,97753957
0.5segs	0,95368951	0,91939349	0,9591658	0,97753957
0.6segs	0,95368951	0,91939349	0,9591658	0,97753957
0.7segs	0,95368951	0,91939349	0,96036175	0,97753957
0.8segs	0,95368951	0,91939349	0,96036175	0,97753957
0.9segs	0,95368951	0,91939349	0,96036175	0,97753957
1segs	0,95368951	0,91939349	0,96036175	0,97753957



Conclusiones

Todas las metaheurísticas alcanzan una buena eficacia, siendo simi la que se queda un poco más atrás que las demás.

ACEi OptTot eleva su eficacia sobre todo en un cierto punto, para mantenerse después casi constante. Algo parecido ocurre con GRASP.

GAG se mantiene constante.

Simi también se mantiene prácticamente constante.

7.1.6. Conclusiones finales

Las metaheurísticas que hemos estudiado e implementado para el TSP, han alcanzado una eficacia lo suficientemente alta para el problema, según los tamaños de las pruebas realizadas. Dicha eficacia supera o se aproxima al 90%, por lo que podríamos decir todas se pueden usar para resolver este problema con una tasa alta de buenos resultados.

Tenemos que destacar que, dentro de estas, la metaheurísticas inspiradas en colonias de hormigas ACEi OptTot y ACEi Opt son las que mejores se adaptan al problema, puesto que como hemos indicado, es una metaheurística apropiada para problemas cuya modelización se puede hacer directamente sobre grafos, además de que hemos incluido la búsqueda local de best improvement usando 2-opt, lo que acelera el proceso. Sin embargo, para que ACEi OptTot alcance la mejor solución posible (incluso el óptimo), requiere de mucho tiempo.

Después de haber realizado pruebas de distinto tipo, y a pesar de que tenemos unas metaheurísticas lentas, las cuales lo son por los respectivos valores de sus parámetros, hemos comprobado que usando una solución inicial y limitando el tiempo de ejecución, se consiguen resultados bastante buenos.

7.2. Mochila 0-1

A continuación, se presentan las comparativas entre los algoritmos utilizados para resolver el problema de la Mochila 0-1.

Para la realización de estas comparativas, se ejecutaron los algoritmos sobre diferentes dimensiones (número de ítems) del problema. Estas instancias de problemas fueron obtenidas de un repositorio de GitHub (https://github.com/patrickherrmann/Knapsack) en forma de ficheros de texto. Cada uno de estos ficheros contiene la representación de los elementos del problema (mochila e ítems). La primera línea consta de un valor que representa el número de ítems del problema. A continuación, se pasa a describir cada uno de estos ítems: cada ítem se describe en una línea nueva y cada línea consta de tres columnas, siendo la primera columna el identificador del ítem, la segunda columna, su beneficio y la tercera columna, su coste. Después de la descripción de los ítems, en una última línea, se describe la capacidad de la mochila del problema.

Aquí presentamos unas cuantas líneas del fichero correspondiente a un problema con 20 ítems:

```
20
    1    91    29
    2    60    65
    3    61    71
    4     9    60
    5    79    45
    6    46    71
    7    19    22
    8    57    97
    9     8     6
   10    84    91
   11    20    57
   12    72    60
   13    32    49
   14    31    89
   15    28     2
   16    81    30
   17    55    90
   18    43    25
   19   100    82
   20    27    19
524
```

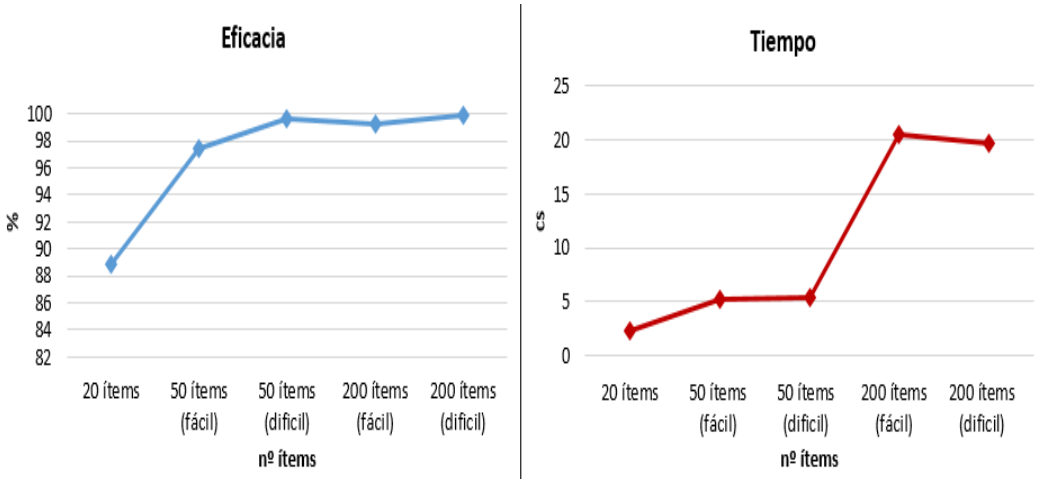
Figura 7.2. Fichero que representa un problema con 20 ítems

7.2.1. Comparativa de eficacia y tiempo según dimensión del problema

El proceso para realizar e interpretar este tipo de comparativa sigue el mismo esquema que su homónimo para el TSP.

ACE Opt (Metaheurística optimizada, inspirada en colonias de hormigas elitistas)

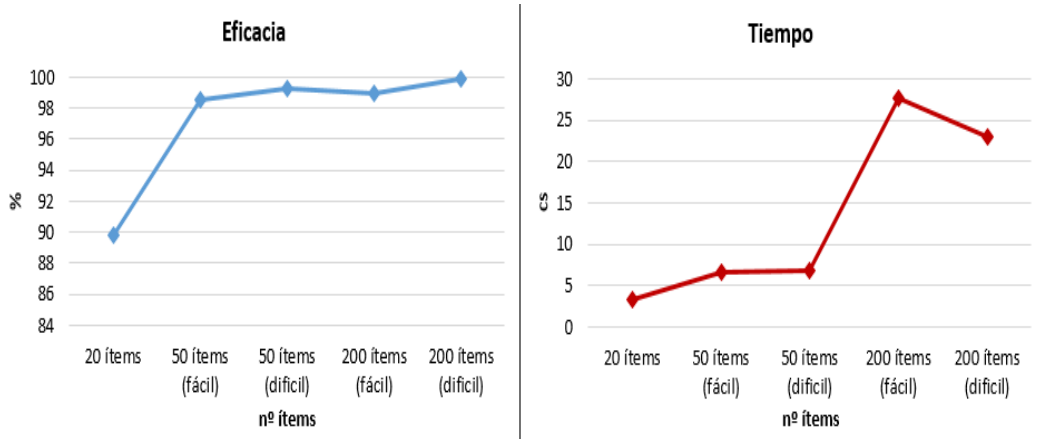
Dimensión	Eficacia	Tiempo
20 ítems	88,84%	2,295cs
50 ítems (fácil)	97,42%	5,18cs
50 ítems (difícil)	99,66%	5,325cs
200 ítems (fácil)	99,29%	20,52cs
200 ítems (difícil)	99,91%	19,685cs



En este algoritmo, solo se realizan búsquedas locales con la mejor hormiga de todas, lo que hace que el tiempo sea menor que con otros algoritmos de este tipo, consiguiendo una eficacia similar.

ACE OptTot (Metaheurística optimizada, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

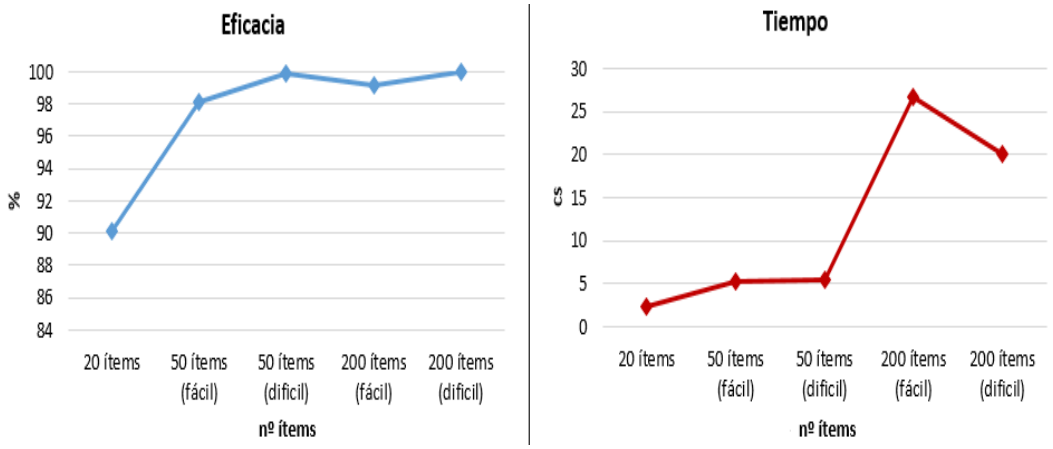
Dimensión	Eficacia	Tiempo
20 ítems	89,81%	3,22cs
50 ítems (fácil)	98,49%	6,585cs
50 ítems (difícil)	99,28%	6,705cs
200 ítems (fácil)	98,97%	27,715cs
200 ítems (difícil)	99,91%	22,92cs



Este algoritmo, por medio de búsquedas locales por parte de cada hormiga después de haber construido su solución, ha dado buenos resultados, por lo que no hizo falta utilizar una solución inicial de partida. No obstante, estas búsquedas incrementaron el tiempo de ejecución.

ACEi Opt (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas)

Dimensión	Eficacia	Tiempo
20 ítems	90,08%	2,285cs
50 ítems (fácil)	98,13%	5,145cs
50 ítems (difícil)	99,86%	5,34cs
200 ítems (fácil)	99,12%	26,595cs
200 ítems (difícil)	99,94%	20,07cs

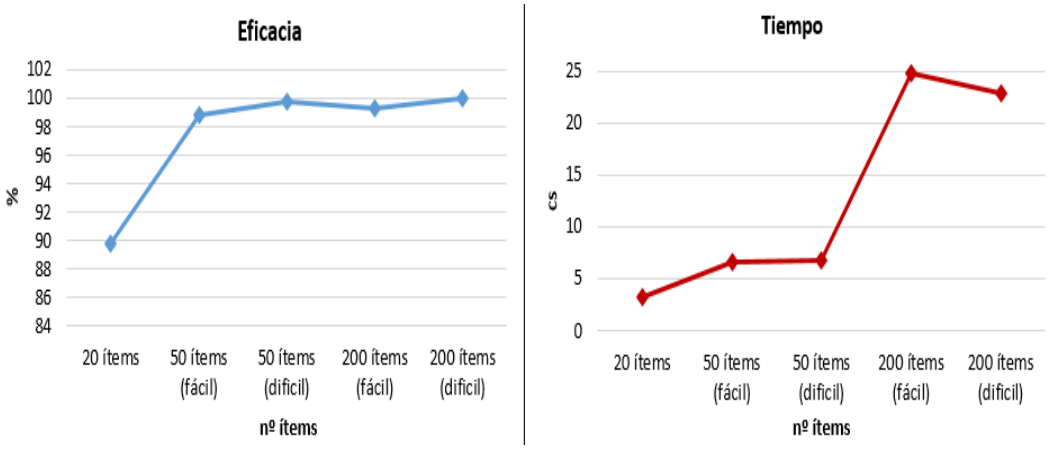


En las distintas instancias del problema, este algoritmo alcanza buenas soluciones.

Al igual que en el TSP, hemos utilizado la implementación elitista, y además hemos empezado con una solución inicial. Esto a veces consigue mejores resultados, pero otras veces no tanto. Esto puede ser debido a que, aunque utilizamos como solución inicial una obtenida mediante la heurística voraz que utiliza el mejor ratio beneficio/costo, en algunos problemas esta no suele llevar a los resultados más prometedores quedándose el algoritmo atascado en óptimos locales.

ACEi OptTot (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

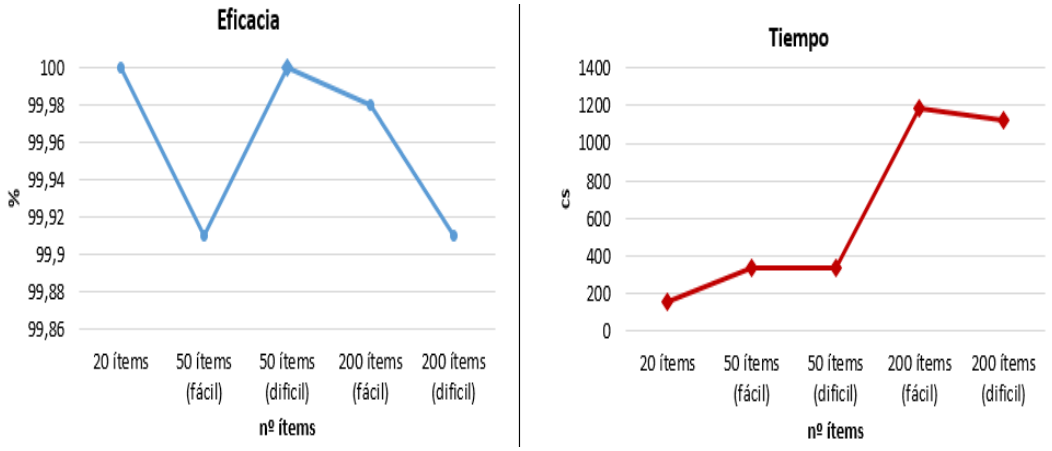
Dimensión	Eficacia	Tiempo
20 ítems	89,81%	3,225cs
50 ítems (fácil)	98,75%	6,58cs
50 ítems (difícil)	99,73	6,715cs
200 ítems (fácil)	99,29%	24,845cs
200 ítems (difícil)	99,92%	22,905cs



Al igual que el algoritmo anterior, usamos una búsqueda local, pero en este caso, la realizamos con todas las hormigas, por lo que, para compensar este proceso costoso en tiempo, hemos reducido los parámetros que lo ralentizan (número de hormigas e iteraciones máximas) para que se acerquen al anterior. Esto ha provocado que su eficacia ronde la del problema anterior.

GAGEsp (Algoritmo genético generacional específico)

Dimensión	Eficacia	Tiempo
20 ítems	100%	158,875cs
50 ítems (fácil)	99,91%	337,43cs
50 ítems (difícil)	100%	336,08cs
200 ítems (fácil)	99,98%	1182,715cs
200 ítems (difícil)	99,91%	1120,355cs



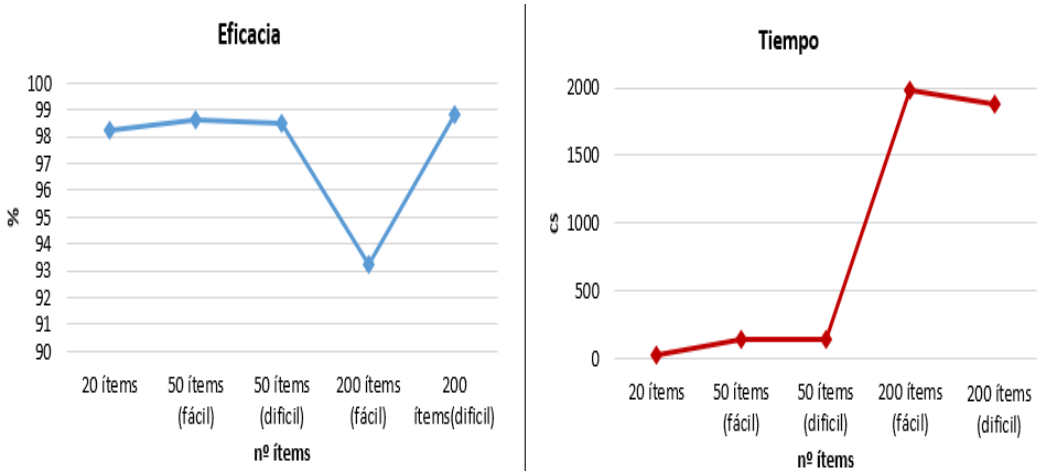
Utilizamos un individuo (solución) creado como solución inicial. El tiempo invertido es aceptable. Para mejorar su eficacia hasta alcanzar la *optimalidad*, hay dos alternativas: subir su número de generaciones (parámetro GENERACIONES), elevando de manera drástica el tiempo, o usar una versión memética del algoritmo. Optamos por la segunda para optimizar. En un principio creíamos que sin la versión memética no era posible alcanzar el óptimo, pero comprobamos que sí.

Además, gracias al uso de una solución inicial, fuimos capaces de lograr una solución final bastante buena. Cabe destacar que es de los algoritmos que más tarda, pero es debido a que lo hemos configurado con unos valores que permiten alcanzar (al menos en los casos estudiados, excepto uno que se queda en 99.95%) la solución óptima.

Al igual que en el TSP, lo parametrizamos como elitista para conseguir una mayor eficacia.

MAGEsp (Algoritmo memético generacional específico)

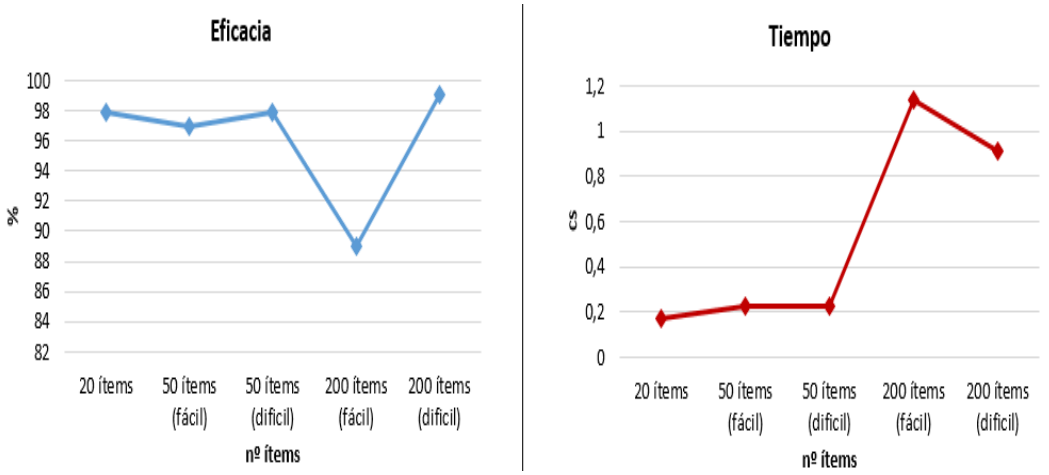
Dimensión	Eficacia	Tiempo
20 ítems	98,21%	28,355cs
50 ítems (fácil)	98,66%	140,93cs
50 ítems (difícil)	98,53%	141,56cs
200 ítems (fácil)	93,23%	1981,195cs
200 ítems (difícil)	98,84%	1181,035cs



Esta versión memética, al tener una búsqueda local, aumenta de manera significativa sus tiempos. La eficacia sigue siendo alta, aunque debido a disminuir los valores de sus parámetros GENERACIONES y TAM_POBLACIÓN (para compensar la búsqueda local), se impide que alcance el óptimo.

GRASP (Procedimiento aleatorio voraz de búsqueda adaptativa)

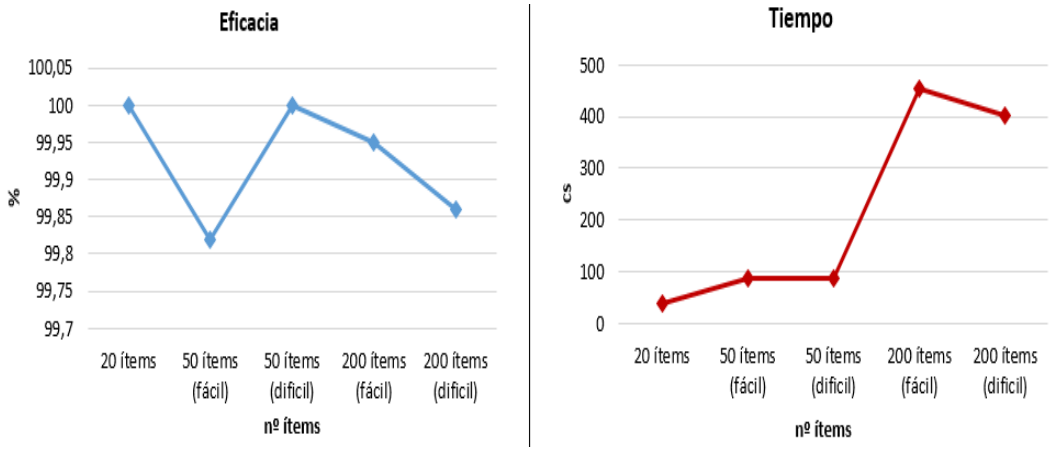
Dimensión	Eficacia	Tiempo
20 ítems	97,93%	0,17cs
50 ítems (fácil)	96,97%	0,225cs
50 ítems (difícil)	97,83%	0,225cs
200 ítems (fácil)	89%	1,135cs
200 ítems (difícil)	99,01%	0,91cs



Con una última optimización, hemos logrado que el algoritmo alcance una alta eficacia con un bajo coste temporal. Anteriormente su eficacia rondaba el 60%, y ahora está en torno al 90%, aunque al ser un poco aleatorio, sus resultados pueden variar.

Simi (Algoritmo de enfriamiento simulado, con solución inicial de partida)

Dimensión	Eficacia	Tiempo
20 ítems	100%	37,425cs
50 ítems (fácil)	99,82%	86,325cs
50 ítems (fácil)	100%	86,375cs
200 ítems (difícil)	99,95%	454,625cs
200 ítems (difícil)	99,86%	403,35cs



En algunos casos, con un tiempo menor que el algoritmo GAGEsp, puede alcanzar una eficacia muy alta, e incluso la óptima. Aunque como hemos comentado, GAGEsp tarda por haber aumentado los valores de sus parámetros para asegurar el óptimo.

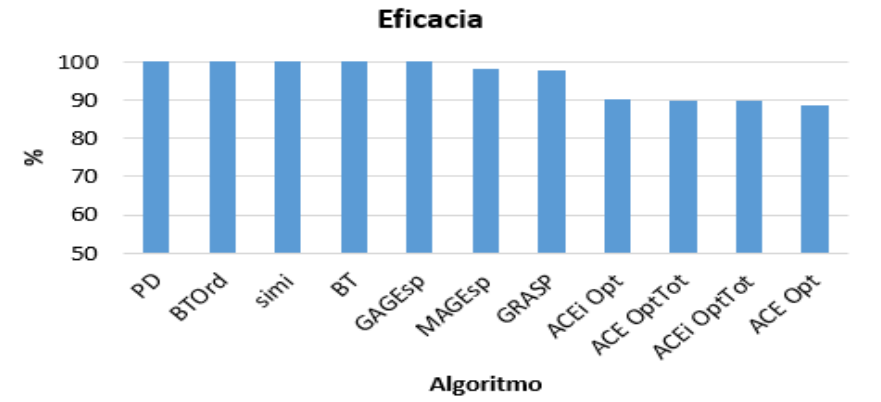
7.2.2. Comparativa conjunta según dimensión del problema

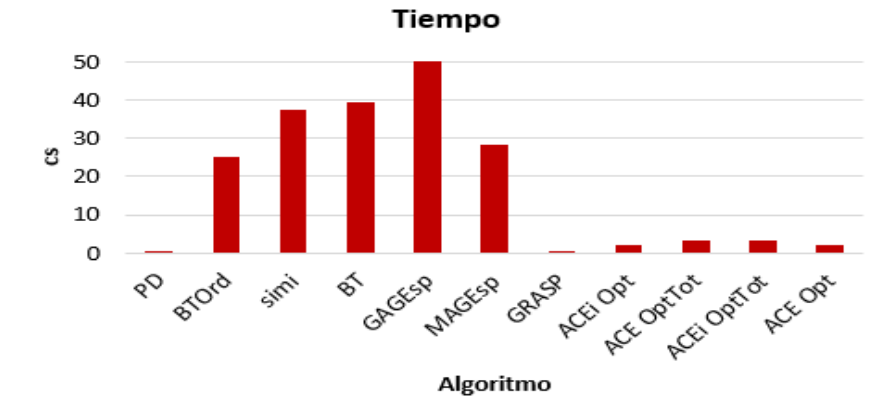
En este tipo de comparativa no existe límite de tiempo para la ejecución de dichos algoritmos, por lo que, se utilizan como valores de comparación la eficacia de la solución obtenida y el tiempo empleado para ello de cada algoritmo y se comparan con los valores correspondientes obtenidos mediante el algoritmo de PD (Programación dinámica), el cual siempre obtiene el óptimo del problema.

A partir de 27 ítems, se deja de utilizar el algoritmo BT (Backtracking), porque su ejecución tardaría demasiado.

20 ítems

Algoritmo	Eficacia %	Beneficio	Coste	T (cs)
PD	100	726	519	0,025
BTOrd	100	726	519	25,005
Simi	100	726	519	37,425
BT	100	726	519	39,37
GAGEsp	100	726	519	158,875
MAGEsp	98,21	713	515	28,355
GRASP	97,93	711	514	0,17
ACEi Opt	90,08	654	523	2,285
ACE OptTot	89,81	652	518	3,22
ACEi OptTot	89,81	652	508	3,225
ACE Opt	88,84	645	519	2,295

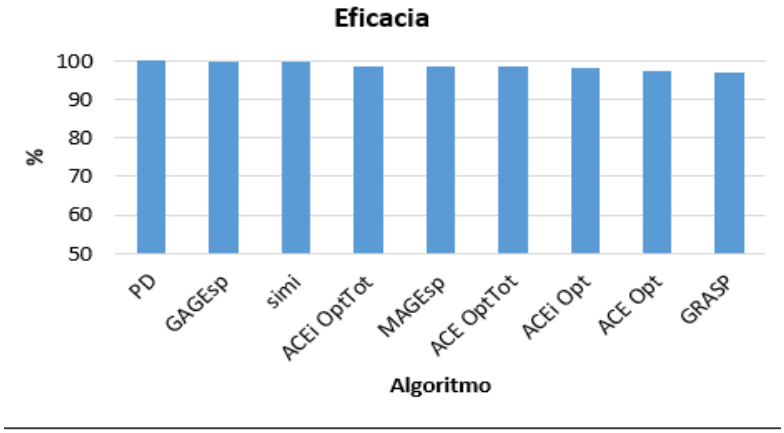


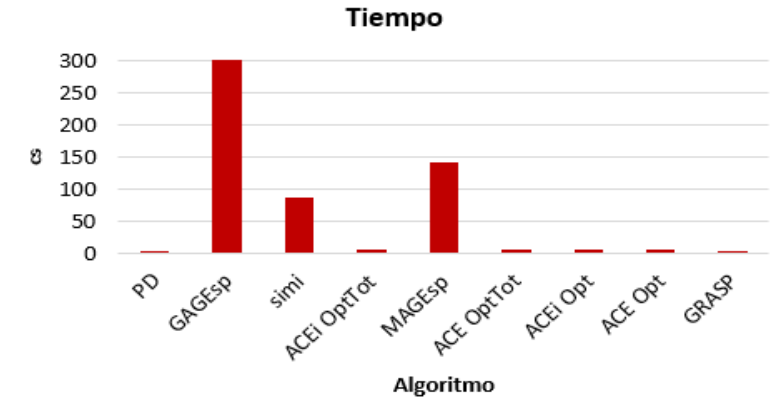


En esta instancia del problema, se puede observar que la eficacia alcanzada por la mayoría de algoritmos es alta. Sin embargo, debemos decir que en la gráfica de tiempos se forman 2 grupos bien diferenciados, uno con algoritmos con tiempos elevados y otro grupo con algoritmos con tiempos cortos.

50 ítems (fácil)

Algoritmo	Eficacia %	Beneficio	Coste	T (cs)
PD	100	1123	250	0,01
GAGEsp	99,91	1122	250	337,43
simi	99,82	1121	250	86,325
ACEi OptTot	98,75	1109	249	6,58
MAGEsp	98,66	1108	249	140,93
ACE OptTot	98,49	1106	249	6,585
ACEi Opt	98,13	1102	249	5,145
ACE Opt	97,42	1094	249	5,18
GRASP	96,97	1089	249	0,225

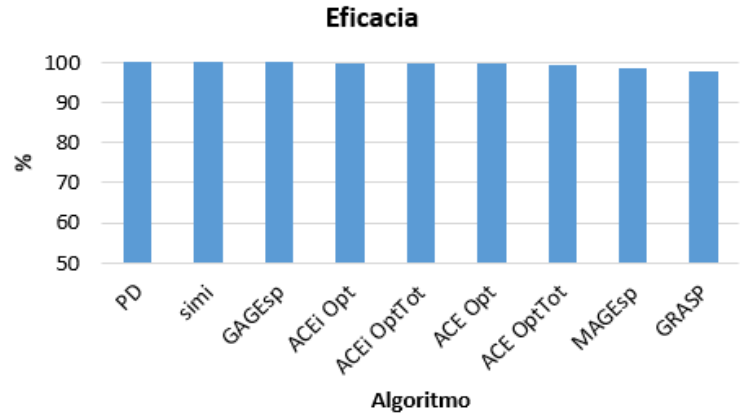


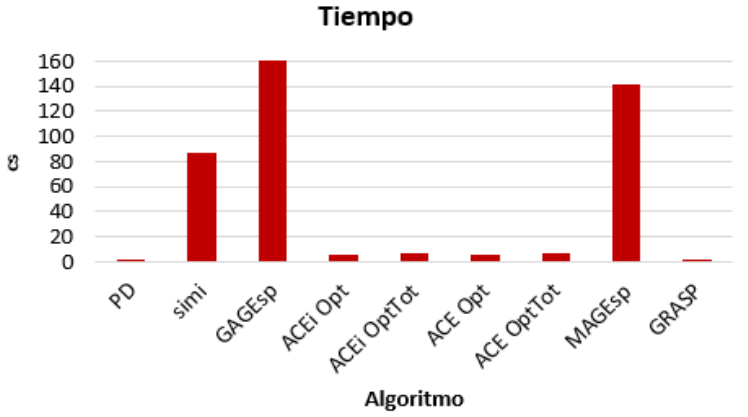


La eficacia de todos los algoritmos es bastante alta. Excepto PD, ninguno consigue el óptimo, pero se aproximan bastante a él. El algoritmo GAGEsp es el que lo consigue a veces pero otras veces no, reduciendo su media. Después de este, está el algoritmo simi, que consigue una eficacia algo menor.

50 ítems (difícil)

Algoritmo	Eficacia %	Beneficio	Coste	T (cs)
PD	100	16610	10110	0,445
simi	100	16610	10110	86,375
GAGEsp	100	16610	10110	336,08
ACEi Opt	99,86	16587	10102	5,34
ACEi OptTot	99,73	16565	10085	6,715
ACE Opt	99,66	16554	10089	5,325
ACE OptTot	99,28	16490	10055	6,705
MAGEsp	98,53	16366	10101	141,56
GRASP	97,83	16250	10090	0,225

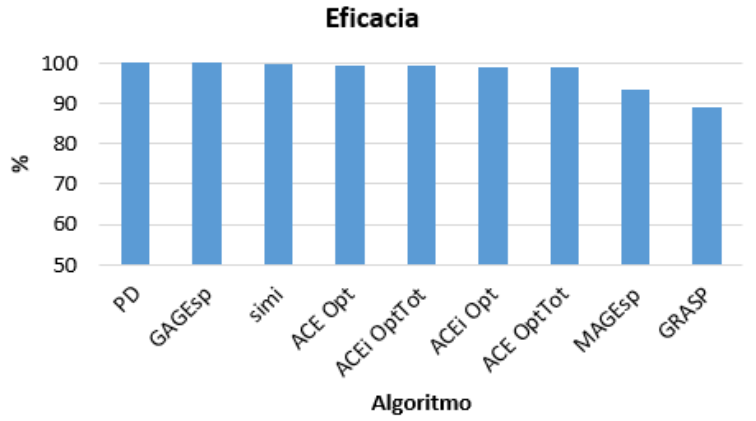


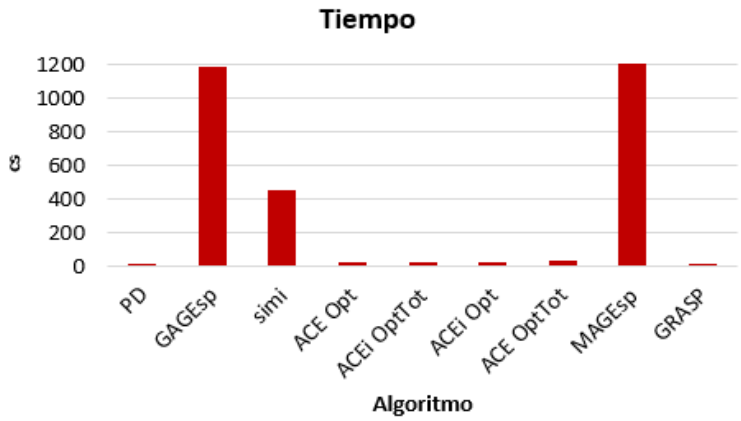


En esta instancia, la eficacia alcanzada por todos los algoritmos ha sido aún mayor que en la anterior, a pesar de aumentar la dimensión del problema.

200 ítems (fácil)

Algoritmo	Eficacia %	Beneficio	Coste	T (cs)
PD	100	4092	2656	0,67
GAGEsp	99,98	4091	2657	1182,715
simi	99,95	4090	2656	454,625
ACE Opt	99,29	4063	2657	20,52
ACEi OptTot	99,29	4063	2655	24,845
ACEi Opt	99,12	4056	2657	26,595
ACE OptTot	98,97	4050	2655	27,715
MAGEsp	93,23	3815	2654	1981,195
GRASP	89	3642	2656	1,135





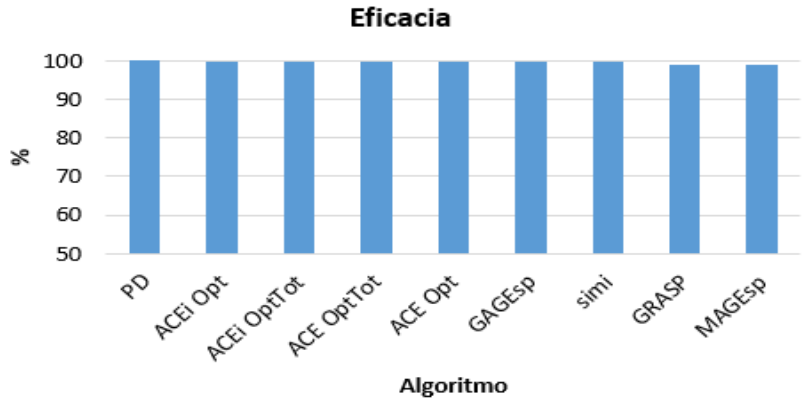
Esta instancia es importante, puesto que en los experimentos que realizamos, la mayoría de veces los algoritmos se quedaban anclados en un máximo local, con eficacia del 99.95%. Por esta razón, decidimos aumentar los parámetros GENERACIONES y TAM_POBLACION del algoritmo GAGEsp, siendo el único que a veces consiguió el 100% de eficacia y otras el 99.95%, lo que ha generado una media de 99.98% (redondeo a 2 decimales).

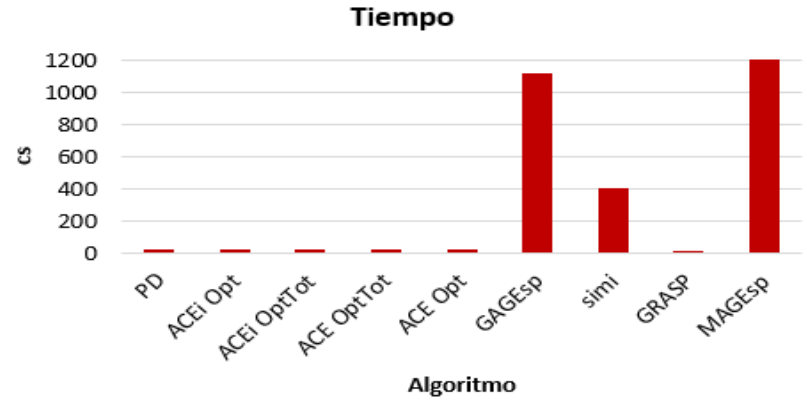
ACEi OptTot ha alcanzado una gran eficacia (aunque no llega a alcanzar a la obtenida por GAGEsp) sin usar mucho tiempo. Esto es debido a que queríamos acelerarlo restringiendo los valores de sus parámetros.

Por lo que respecta al resto de algoritmos, estos han conseguido buenas soluciones en tiempos razonables, siendo el algoritmo GRASP, el que se ha quedado más atrás en cuanto a la eficacia de su solución obtenida.

200 ítems (difícil)

Algoritmo	Eficacia %	Beneficio	Coste	T (cs)
PD	100	137448	112648	18,305
ACEi Opt	99,94	137371	112621	20,07
ACEi OptTot	99,92	137334	112594	22,905
ACE OptTot	99,91	137331	112611	22,92
ACE Opt	99,91	137320	112620	19,685
GAGEsp	99,91	137319	112644	1120,355
simi	99,86	137257	112647	403,35
GRASP	99,01	136081	112586	0,91
MAGEsp	98,84	135856	112641	1881,035





A pesar de que esta es una instancia grande del problema, la eficacia obtenida por todos los algoritmos ha sido buena, no bajando esta del 98%.

Conclusiones

Como podemos comprobar, en este tipo de comparativas, sin necesidad de *memetismo*, GAGEsp puede a veces alcanzar el óptimo, si aumentamos los valores de sus parámetros. Si se baja el número de iteraciones se ha comprobado que su eficacia queda como la del resto de los algoritmos, no bajando del 99%.
El algoritmo memético MAGEsp, alcanza una gran eficacia, tardando más o menos lo mismo que GAGEsp, al haber reducido los valores de sus parámetros GENERACIONES y TAM_POBLACIÓN.

Las metaheurística inspiradas en colonias de hormigas (ACE Opt, ACE OptTot, ACEi Opt y ACEi OptTot), han conseguido buenos resultados en tiempos aceptables.

La metaheurística simi ha dado buenos resultados, pero sus tiempos no han sido del todo buenos.

GRASP, gastando aún menos tiempo que simi, ha conseguido una buena eficacia, sin llegar a superarlo, no quedándose tan lejos de este.

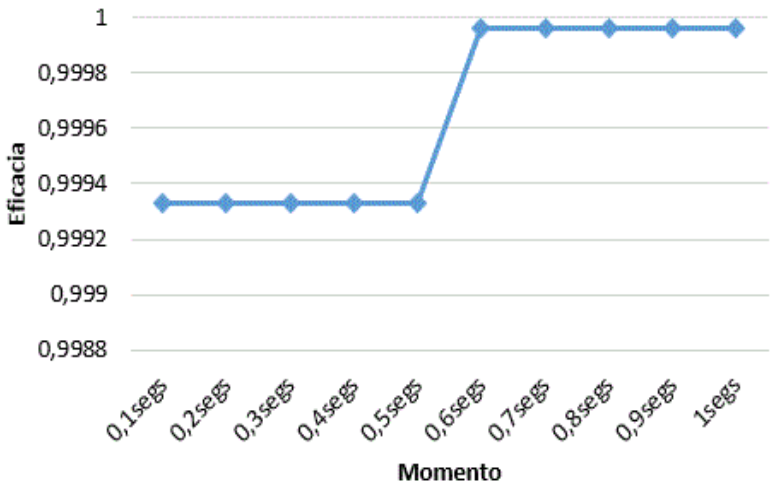
7.2.3. Progresión de la evolución de la eficacia (individual)

El proceso para realizar e interpretar este tipo de comparativa sigue el mismo esquema que su homónimo para el TSP.

200 ítems (difícil)

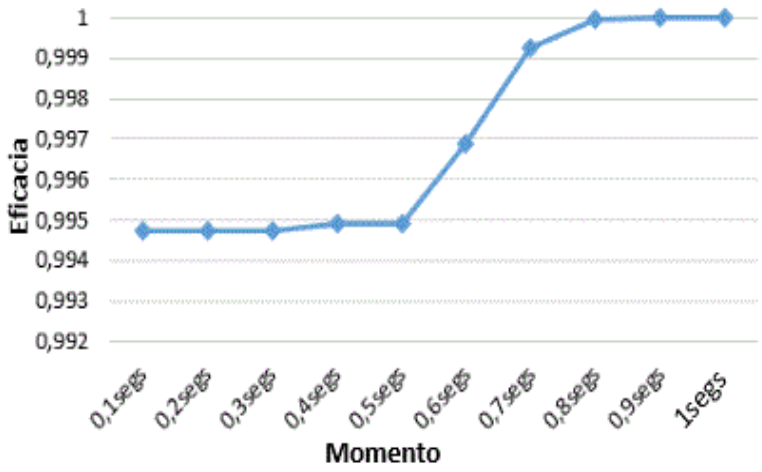
ACEi OptTot (Metaheurística optimizada, con solución inicial de partida, inspirada en colonias de hormigas elitistas, con búsqueda local por cada hormiga después de haber construido su solución)

Momento	Eficacia
0,1segs	0,99933066
0,2segs	0,99933066
0,3segs	0,99933066
0,4segs	0,99933066
0,5segs	0,99933066
0,6segs	0,99996362
0,7segs	0,99996362
0,8segs	0,99996362
0,9segs	0,99996362
1segs	0,99996362



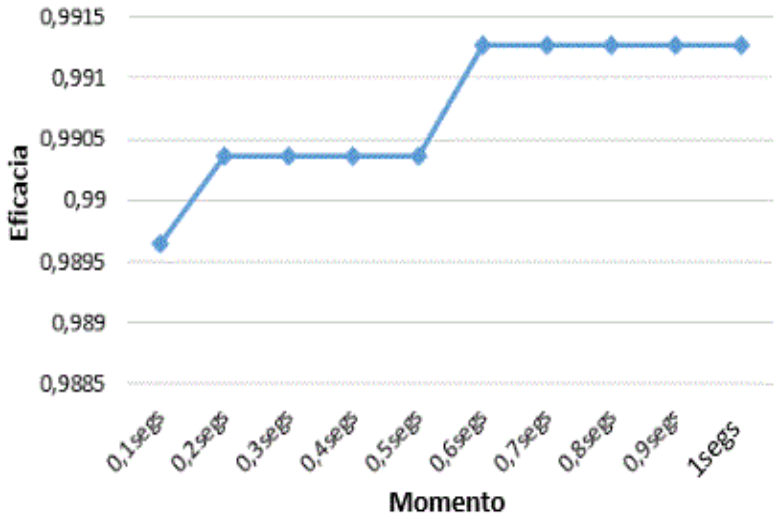
GAGEsp (Algoritmo genético generacional específico)

Momento	Eficacia
0,1segs	0,99473255
0,2segs	0,99473255
0,3segs	0,99473255
0,4segs	0,99491444
0,5segs	0,99491444
0,6segs	0,99690065
0,7segs	0,99927245
0,8segs	0,9999709
0,9segs	1
1segs	1



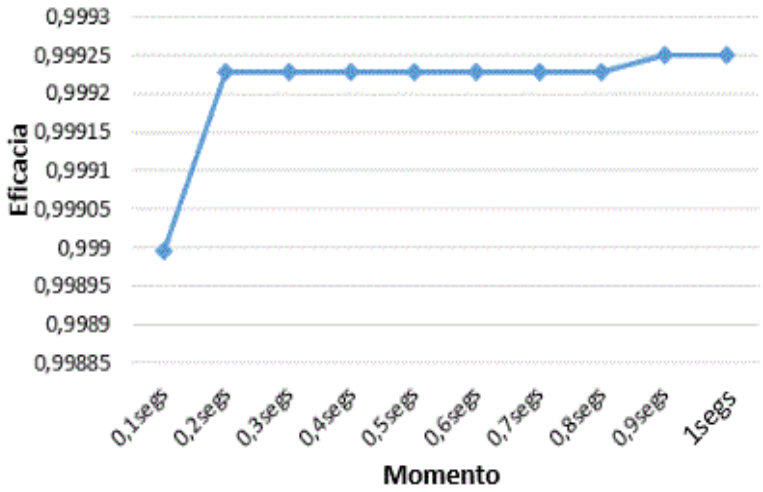
GRASP (Procedimiento aleatorio voraz de búsqueda adaptativa)

Momento	Eficacia
0,1segs	0,98963972
0,2segs	0,99036727
0,3segs	0,99036727
0,4segs	0,99036727
0,5segs	0,99036727
0,6segs	0,99126943
0,7segs	0,99126943
0,8segs	0,99126943
0,9segs	0,99126943
1segs	0,99126943



simi (Algoritmo de enfriamiento simulado, con solución inicial de partida)

Momento	Eficacia
0,1segs	0,99899598
0,2segs	0,9992288
0,3segs	0,9992288
0,4segs	0,9992288
0,5segs	0,9992288
0,6segs	0,9992288
0,7segs	0,9992288
0,8segs	0,9992288
0,9segs	0,99925063
1segs	0,99925063



Conclusiones

ACEi OptTot hace un crecimiento en la mitad y después se mantiene constante su eficacia.

GAGEsp crece de manera curiosa, pero donde más crece es después de la quinta décima de segundo.

Grasp, debido a su aleatoriedad, consigue subidas cada cierto tiempo. En este caso, en dos puntos, en la primera y la quinta décima de segundo.

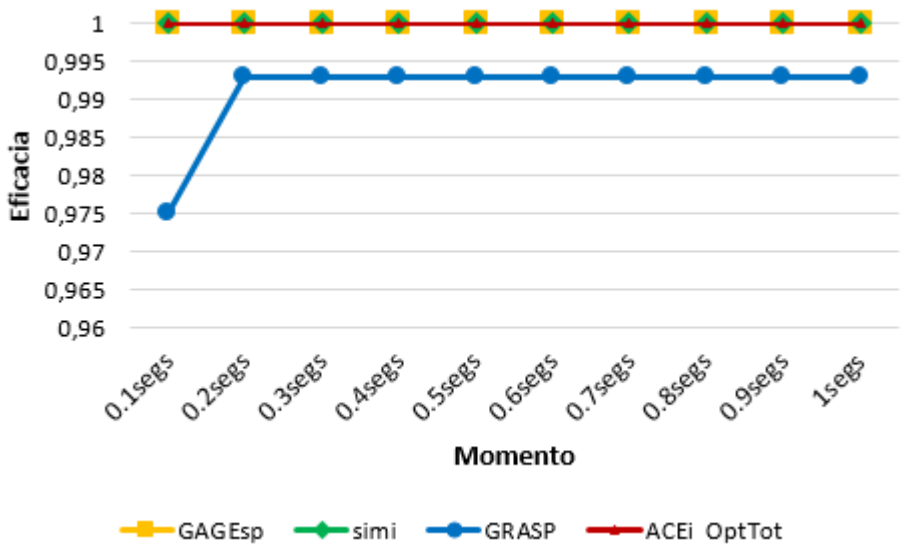
Simi consigue una mejora en la primera décima de segundo, pero después se mantiene prácticamente constante.

7.2.4. Progresión de la evolución de la eficacia (todos los algoritmos sobre una misma gráfica)

El proceso para realizar e interpretar este tipo de comparativa sigue el mismo esquema que su homónimo para el TSP.

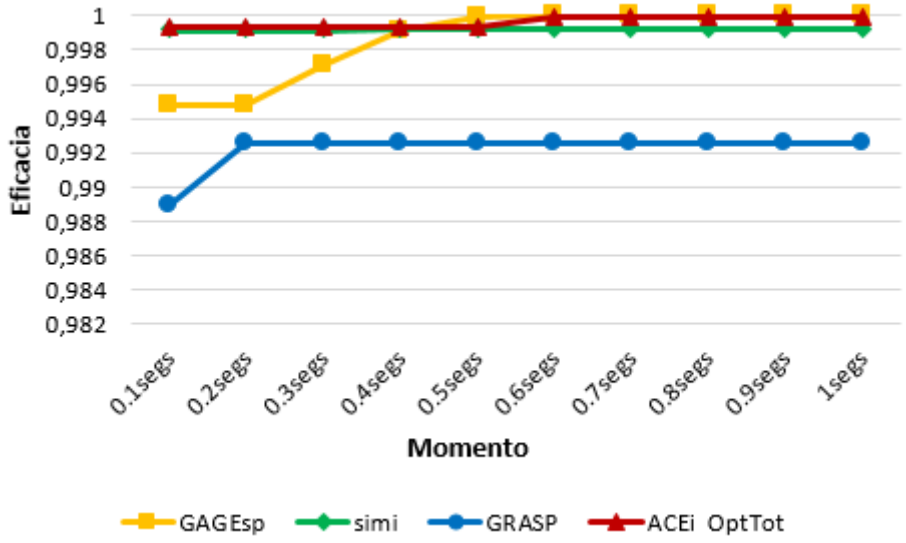
50 ítems (difícil)

Momento	GAGEsp	simi	GRASP	ACEi OptTot
0.1segs	1	1	0,97507526	1
0.2segs	1	1	0,99307646	1
0.3segs	1	1	0,99307646	1
0.4segs	1	1	0,99307646	1
0.5segs	1	1	0,99307646	1
0.6segs	1	1	0,99307646	1
0.7segs	1	1	0,99307646	1
0.8segs	1	1	0,99307646	1
0.9segs	1	1	0,99307646	1
1segs	1	1	0,99307646	1



200 ítems (difícil)

Momento	GAGEsp	simi	GRASP	ACEi OptTot
0.1segs	0,99473255	0,9991706	0,98890489	0,99933066
0.2segs	0,99482713	0,9991706	0,99257174	0,99933066
0.3segs	0,99707526	0,9991706	0,99257174	0,99933066
0.4segs	0,99910512	0,99924335	0,99257174	0,99933066
0.5segs	0,99995635	0,99924335	0,99257174	0,99933066
0.6segs	0,99998545	0,99924335	0,99257174	0,99996362
0.7segs	1	0,99924335	0,99257174	0,99996362
0.8segs	1	0,99924335	0,99257174	0,99996362
0.9segs	1	0,99924335	0,99257174	0,99996362
1segs	1	0,99924335	0,99257174	0,99996362



Conclusiones

Como vemos, todas las metaheurísticas alcanzan una gran eficacia comparándolos con el algoritmo de PD (Programación dinámica).

Con menos dimensión tanto la metaheurística GAGEsp como simi, han conseguido la *optimalidad*. Por otra parte, ACEi OptTot no eleva mucho su eficacia, y GRASP experimenta un crecimiento, pero al llegar a cierto punto, dicho crecimiento se ralentiza de tal manera que parece que no varía.

7.2.5. Conclusiones finales

Al igual que en el TSP, todas las metaheurísticas se adaptan bien al problema de la Mochila 0-1, con una eficacia y tiempo razonables.

Los algoritmos genéticos GAGEsp son los que mejor se adaptan, puesto que casi siempre alcanzan la solución óptima, tomando como referencia la solución óptima obtenida mediante el algoritmo de programación dinámica. Para alcanzar la *optimalidad*, hemos tenido que incrementar los valores de los parámetros relacionados con el número de individuos, estos son, el número de generaciones y el número de individuos por generación. Dicho aumento, ha provocado un gran incremento en el tiempo. Aunque sin este aumento, es posible alcanzar la *optimalidad* en muchos casos con el menor tiempo, siendo el mejor en clasificado en el ratio eficacia/tiempo. En otras instancias del problema, como, por ejemplo, en la instancia 200 (fácil), puede quedarse atascado en un máximo local.

MAGEsp es un algoritmo bastante bueno que alcanza resultados muy razonables. Para ello hemos utilizado un individuo (solución) creado como solución inicial, y búsqueda local. El tiempo invertido no es tan grande, gracias a la reducción del valor de sus parámetros, aunque exista una diferencia de un problema a otro. Para mejorar su eficacia, lo parametrizamos utilizando la versión elitista.

ACEi OptTot es una metaheurística bastante buena, aunque puede ser algo lenta, pero sin ser el tiempo invertido, tan elevado para este problema de la Mochila 0-1. Para mejorar su eficacia usamos una versión elitista.
Para reducir el tiempo podemos usar la versión ACEi Opt.

GRASP es una metaheurística bastante buena, aunque para intentar mejorarla, la hemos ralentizado aumentando su número de ciclos.

Simi es una metaheurística buena y es bastante rápida. Ha conseguido llegar al óptimo en diversas ocasiones con problemas de dimensión pequeña. Para conseguir su eficacia se ha utilizado una solución inicial. En caso de no usarla, su eficacia en cuanto a obtener buenas soluciones puede ser peor, aunque su rapidez no disminuye tanto. El uso de una solución inicial, en ocasiones ha provocado una temprana convergencia de la metaheurística.

7.3. Conclusiones finales del capítulo

El objetivo principal de este TFG, es el estudio y comparación de las metaheurísticas estudiadas. Para ello hemos realizado estudios de todo tipo. Después de esto, hemos comprobado que la eficacia de los algoritmos es buena, y que, aunque para finalizar requieran de tiempo, a veces no es necesario gastar tanto para llegar a una solución buena.

Tenemos que destacar que los algoritmos genéticos GAGEsp son los que mejor se adaptan, puesto que siempre alcanza una solución bastante buena, entre el 90 y el 100%. Para alcanzar la *optimalidad*, hemos tenido que incrementar los valores de sus parámetros relacionados con el número de individuos (soluciones). Estos parámetros son, el número de generaciones y el número de individuos por generación. Dicho aumento, ha provocado un gran incremento en el tiempo. Aunque sin este aumento, es posible alcanzar la *optimalidad* en muchos casos con el menor tiempo. Esto sobre todo ha sido necesario para el problema de la Mochila 0-1. Además, siempre, el algoritmo GAGEsp se puede convertir en memético (MAGEsp), utilizando una búsqueda local, lo que mejora su eficacia, aunque aumente el tiempo. Para ello, como es lógico tenemos que disminuir el valor de los parámetros para que no use demasiado tiempo. Además, debido a que disponemos de un framework, su adaptación a distintos tipos de problemas es bastante fácil, lo que consideramos un punto a favor.

Las metaheurísticas inspiradas en colonias de hormigas, son bastante buenas, aunque como esperábamos, en el TSP que es un problema modelado como grafos, obtienen mejores resultados. Para incrementar la eficacia hemos utilizado una búsqueda local.

Simulated annealing suele dar una alta eficacia en un tiempo muy reducido. En el problema de la Mochila 0-1 incluso alcanza en ciertos casos el óptimo y en el TSP, consigue buenos resultados.

GRASP es un algoritmo bastante bueno, aunque a veces puede ser lento, cuya eficacia aumenta en el tiempo.

"Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, y la otra es hacerlo tan complejo que no haya deficiencias obvias", C.A.R. Hoare.

Capítulo 8

Tecnologías utilizadas y sistema

En este capítulo presentaremos las diferentes tecnologías utilizadas, hablaremos sobre la arquitectura sobre la que hemos desarrollado nuestro trabajo y sobre la implementación de los algoritmos estudiados.

8.1. Tecnologías utilizadas

8.1.1. Java

El nacimiento de Java supuso un gran avance en el mundo del software, sobre todo por traer consigo características fundamentales que marcaron la diferencia entre este lenguaje y los existentes en aquel momento. Aunque no vamos a mencionar todas ellas, sí que vamos a exponer algunas que nos parecen interesantes y que han facilitado el desarrollo de nuestro trabajo.

1. Ofrece toda la potencia del paradigma de programación orientado a objetos con una sintaxis fácilmente comprensible y accesible, y un entorno robusto.

2. Pone al alcance de cualquiera la utilización de aplicaciones. Una aplicación escrita en Java se puede ejecutar en cualquier plataforma.

3. Proporciona un conjunto de clases ya implementadas que son potentes, flexibles, y de gran utilidad para los desarrolladores. Estas clases contienen, por ejemplo, estructuras de datos realmente útiles, métodos para manipulación de ficheros, métodos de conexión a bases de datos, un conjunto de protocolos de internet, entre muchas otras funcionalidades.

4. Las clases Java se organizan jerárquicamente, de modo que se generan árboles de herencia. La herencia permite que una clase hija pueda tomar propiedades de su clase padre. Esto deriva en la simplificación de los diseños, evitando escribir código repetido.

5. Otra característica fundamental es el concepto de polimorfismo. Esta palabra viene a expresar las múltiples formas de comportamiento que puede adoptar un objeto, dependiendo de su contexto. Por ejemplo, un método de cualquier clase de objeto puede presentar diferentes implementaciones en función de los argumentos que recibe, recibir diferentes números de parámetros para realizar una misma operación, y realizar diferentes acciones dependiendo del nivel de abstracción en que sea utilizado.

Todas estas son razones suficientes, por las que decidimos elegir Java para la implementación de las estructuras de datos que modelizan los problemas seleccionados y los algoritmos que los resuelven. También fue el lenguaje utilizado para implementar los servicios web.

8.1.2. Tecnologías web

8.1.2.1. HTML5

HTML (Hyper Text Markup Language) es un lenguaje de marcado, que no puede ser considerado en ningún caso un lenguaje de programación, pues su propósito es indicar a los navegadores web como han de estructurar y presentar la información. Para esto cuenta con un conjunto de etiquetas, cada una con un propósito. La sintaxis de la utilización de estas etiquetas se define de la siguiente manera:

<nombre etiqueta>contenido</nombre etiqueta>

Además, para cada etiqueta, podemos realizar una configuración mediante los atributos de las mismas. Así, por ejemplo, las etiquetas suelen contar con los atributos "id" y "name", las cuales sirven para identificarlas dentro del fichero HTML.

El contenido de todo fichero HTML debe estar encerrado entre las etiquetas <HTML></HTML> y este consta de dos partes principales desarrolladas dentro de las etiquetas <HEAD> Y <BODY>:

- <HEAD>: incluye la información relativa a la página, como el título, características, codificación, versiones, autores… Además, incluye los enlaces a otro fichero necesario, como scripts JavaScript o CSS.

- <BODY>: define el cuerpo del fichero HTML e incluye los elementos de la página web encargados de la estructuración y presentación de la información. Estos elementos pueden ser: textos, enlaces, tablas, diverso contenido multimedia, etc.

Para el desarrollo de la parte de presentación de nuestro cliente web, hemos usado la quinta versión de HTML (HTML5), la cual está estandarizada por la WC3 (World Wide Web Consortium) y que incluye nuevas características que hacen más rápido y versátil la construcción de contenido web. Algunas de estas características son:

1. Es nativo y no pertenece a nadie (open source).

2. Ofrece una mayor compatibilidad con los navegadores de dispositivos móviles tan usados en nuestros días.

3. Ofrece la inclusión directa de contenido multimedia como audio y vídeo.

4. Ofrece soporte para codecs específicos.

5. No estar conectado, no le impide poder ejecutar páginas web.

6. Permite adaptar un mismo diseño a distintos dispositivos como tablets o teléfonos móviles.

8.1.2.2. CSS3

CSS3es la tercera generación de CSS (Cascade Style Sheet), y también está estandarizado por W3C.los ficheros .css definen el estilo y apariencia de los contenidos web.
Permite crear proyectos web de manera eficiente, obteniendo resultados más flexibles y utilizables. CSS3 no es una nueva especificación de CSS, si no que se ha decantado por una filosofía de módulos independientes que agrupa diferentes elementos del lenguaje que no dependen unos de otros, de modo que cada uno de ellos puede ser desarrollado de manera independiente.

Recordemos que CSS3 presenta novedades sobre este propósito, como la creación de sombras, esquinas redondeadas, gradientes, fondos semitransparentes y otros muchos efectos visuales.

Al utilizar esta tecnología en nuestro cliente web, siempre que lo vimos posible, intentamos separar el contenido HTML de su apariencia. Para ello codificamos las css en ficheros aparte. De este modo un mismo fichero HTML podía incluir más de un fichero CSS, cuya combinación diese como resultado una apariencia final. Además, este modo de organización, nos facilitó la realización de cambios en cuanto a la apariencia de nuestras páginas HTML, ya que solo bastaba con modificar el o los ficheros CSS en cuestión para que los cambios se viesen reflejados automáticamente en todas las páginas en donde estos estuviesen referenciados.

8.1.2.3. Materialize

Es un framework front-end basado en Material-Desing, para la creación de interfaces web. Ofrece diseño *responsivo*. Nos ha sido de gran ayuda para la mejora visual de nuestras interfaces web.

8.1.2.4. JavaScript

Si nos remontamos a las primeras páginas web de la historia de la World Wide Web, el contenido de estas estaba básicamente restringido a textos y enlaces. Hoy en día las páginas web están llenas de contenido multimedia, que incluyen audio, video, imágenes, gráficos animados, efectos, navegación interactiva, donde en la mayor parte, la tecnología JavaScript juega un papel fundamental.

JavaScript es un lenguaje de programación interpretado, lo que quiere decir que el código fuente generado no requiere ser previamente compilado para ser ejecutado. Cada fichero de código fuente contiene un guion (script) que se ejecutará en la parte del cliente. Estos scripts pueden acceder a los elementos HTML, que por sí solos son estáticos, pero que, gracias a JavaScript, pueden tomar vida, ofreciendo una experiencia mucho más interactiva a los usuarios.

Aunque el desarrollo de JavaScript surgió de la mano del antiguo navegador de Netscape, Netscape Navigator 3.0, actualmente todos los navegadores web son capaces de interpretarlo.

La dinámica de trabajo de JavaScript está basada en la manipulación del DOM (Document Object Model). El DOM es una API para documentos HTML y XML, donde viene definida la estructura lógica de cada uno de ellos y el modo en que estos son accedidos y manipulados. Gracias a esto, los desarrolladores mediante JavaScript, pueden navegar a través de documentos HTML o XML (Extensible Markup Language), añadiendo, modificando o eliminando elementos y contenido.

DOM está representado en forma de árbol. Un ejemplo para una página HTML simple es el de la siguiente figura:

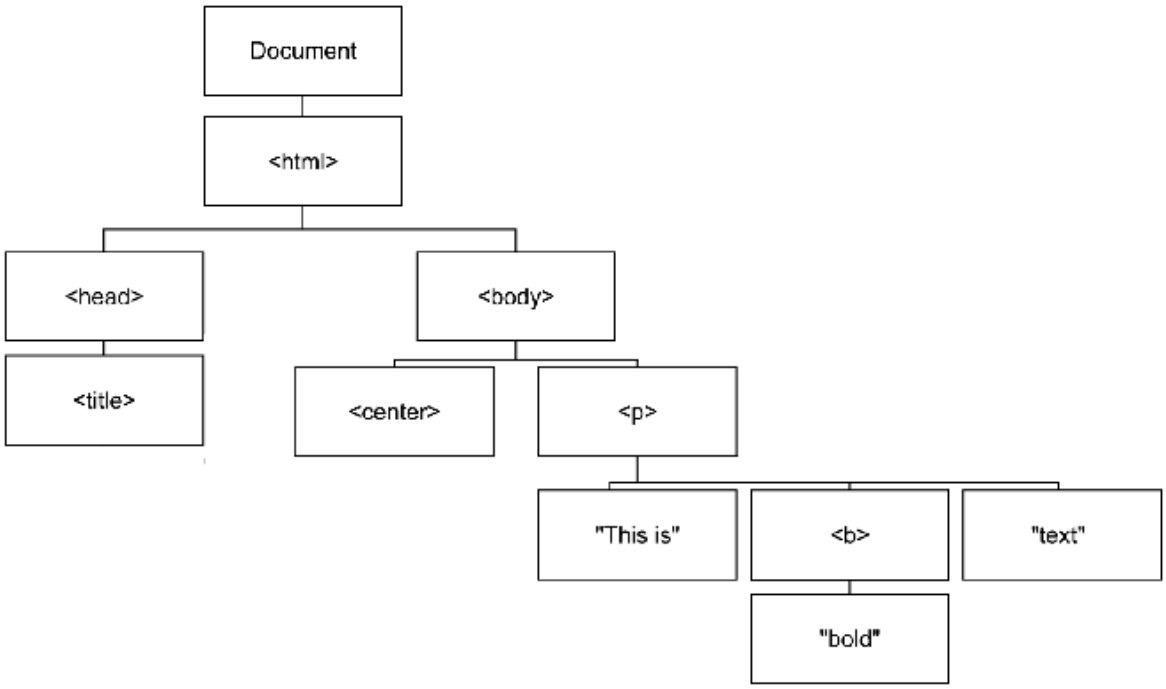


Figura8.1. DOM para una página HTML básica

8.1.2.5. JQuery

Es una librería desarrollada en JavaScript, por tanto, también se basa en la manipulación del DOM y cuyo lema fundamental es *"permitir hacer mucho más escribiendo muchísimo menos código"*. JQuery surge hace años, cuando cada navegador interpretaba de manera distinta el código JavaScript, lo que traía grandes problemas a los desarrolladores. Estos tenían que adaptar el código a cada navegador, lo que requería un conocimiento previo y en profundidad de las particularidades de cada uno con el fin de saber en qué se diferenciaban y poder escribir el código adecuado.
Con JQuery desaparecen estos problemas, permitiendo utilizar todas las características de JavaScript, aún incluso sin tener un amplio conocimiento de este, lo que conlleva al enriqueciendo cada vez más notable del contenido web. Con JQuery se pueden manejar eventos de manera más sencilla, generar efectos de todo tipo, animaciones, realizar peticiones AJAX (Asynchronous JavaScript And XML), etc.; y todo esto escribiendo mucho menos cantidad de líneas de código que con JavaScript puro.

JQuery ha jugado un papel importante en la parte del cliente web de nuestro proyecto, ya que nos facilitó el manejo de eventos y el trabajo con la API de Google Maps, de la cual hablaremos también más adelante.

8.1.2.6. AJAX

AJAX (Asynchronous JavaScript And XML) es una técnica de desarrollo web para crear aplicaciones interactivas. Desde un punto de vista práctico, es una tecnología que sirve para solicitar contenido al servidor y mostrarlo sin que sea necesario actualizar completamente la parte del cliente. Las aplicaciones que hacen uso de AJAX se ejecutan en el navegador web, mientras en segundo plano se sigue manteniendo una comunicación asíncrona con el servidor.

AJAX está presente en muchos sitios web, tales como Google Maps, Gmail, Facebook, Twitter, entre muchos otros.

Utilizar AJAX en nuestro proyecto fue de suma importancia ya que necesitábamos encontrar alguna técnica con la que realizar peticiones a nuestro servidor para poder consumir ciertos servicios web de forma asíncrona.

8.1.2.7. PHP

PHP (Pre Hypertext-processor) es un lenguaje de programación del lado del servidor, adecuado para generar contenido web dinámico y de código abierto. Fue uno de los primeros lenguajes que se podían incrustar directamente en documentos HTML. Es de carácter general y permite trabajar bajo el paradigma orientado a objeto, es de *tipado* débil (la comprobación de los tipos se hace en tiempo de ejecución), además, cuenta con herencia (simple, no múltiple) y tiene

El código PHP puede ser ejecutado en distintos Sistemas Operativos, como los basados en UNIX (Linux, Mac OS, etc), o Microsoft Windows. Tiene múltiples funcionalidades como la gestión de formularios o el acceso a bases de datos y proporciona estructuras de datos muy útiles.

Su fama actual es tal que es utilizado en millones de sitios web y servidores en todo el mundo y ejemplo de ello son sitios tan conocidos como Wikipedia o Facebook.

Aunque en nuestro proyecto no hicimos mucho uso de él, nos ha servido para incluir partes de código en el cliente web tales como cabeceras prediseñadas, listas de elementos y parámetros para los algoritmos, entre otras.

8.1.2.8. JSON

JSON (JavaScript Object Notation) Es un formato para el intercambio de datos. Básicamente JSON describe los datos con una sintaxis dedicada que se usa para identificar y gestionar los datos. Nació como una alternativa a XML, y su fácil uso en JavaScript ha generado un gran número de seguidores de esta tecnología. Una de las mayores ventajas que tiene el uso de JSON es que puede ser leído por cualquier lenguaje de programación, por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías.

Lo hemos utilizado para recibir e interpretar los datos relativos a distancias entre dos puntos proporcionados la API de Google Maps.

8.1.3. Servicios web

Los servicios web son básicamente unos marcos de interacción entre un cliente y un servidor. En este contexto, se establecen y documentan unos requisitos de uso de estos servicios.
La gran ventaja de los servicios web es que son totalmente independientes de lenguajes de programación y de plataformas. Por ejemplo, un servicio web desarrollado en java, puede ser usado por clientes escritos en otros lenguajes y viceversa. Para que esto sea posible, los servicios web cuentan con un conjunto de protocolos de comunicación, que se exponen a continuación.

8.1.3.1. Protocolos de servicios web

Los servicios web pueden ser consumidos mediante protocolos estándar de comunicación, los cuales pueden ser clasificados según la función que desempeñen:

- Directorio de servicios web: definen dónde se describen y localizan los servicios en la red, facilitando su acceso. UDDI (Universal Description, Discovery and Integration) es el protocolo que normalmente se utiliza para este fin.

- Descripción de servicios web: describe la interfaz pública de funcionamiento de los servicios web. El formato WSDL (Web Service Description Language) es el utilizado normalmente para realizar esta función. WSDL es un documento en formato XML (Extensible Markup Language), que facilita entender, describir y mantener servicios.

- Invocación de métodos de servicios web: para la codificación de los mensajes en formato XML, de modo que puedan ser entendidos extremo a extremo dentro de una conexión en red. Existen protocolos como SOAP (Simple Object Access Protocol), XML-RPC (XML-Remote Procedure Call) o REST (Representational State Transfer) que actualmente son utilizados para cubrir de esta función.
- Servicio de transporte: mediante el cual se realiza el transporte de mensajes entre las aplicaciones de red y protocolos tales como HTTP (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol).

Todos estos protocolos forman parte de la llamada *Pila de protocolos para servicios web*, que se muestra en la siguiente figura:

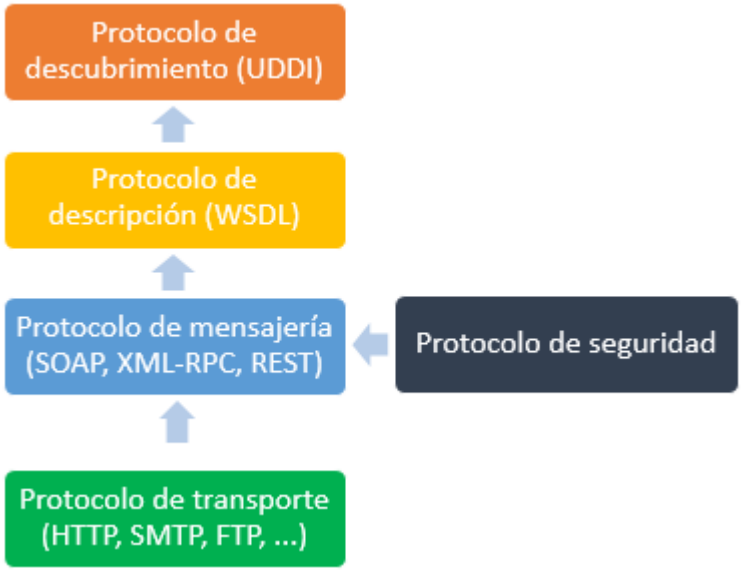


Figura 8.2. Pila de protocolos para servicios web

Esta tecnología de servicios web es una parte importante de nuestro trabajo, pues desarrollamos los servicios necesarios para ofrecer la resolución mediante distintos métodos algorítmicos de los problemas del TSP y la Mochila 0-1. Estos servicios escritos en Java están alojados en nuestro servidor local y son consumidos por nuestro cliente web mediante JavaScript. Para ello utilizamos UDDI, SOAP, WSDL.

8.1.3.2. JAX-WS

Al crear nuestros servicios web con Java, nos decantamos por que utilizar JAX-WS, que es una API del propio Java que facilita la implementación e invocación de estos servicios.
Cuando se desarrollan servicios web, independientemente de cual sea la tecnología utilizada para ello, podemos hacer uso de dos enfoques de trabajo distintos: por un lado, podemos crear los servicios web mediante un WSDL ya existente. A este enfoque se le denomina descendente o top-down. Y, por otro lado, podemos hacer uso de un enfoque ascendente o bottom-up para generar el WSDL mediante código fuente ya existente. JAX-WS se basa precisamente este último enfoque bottom-up.

JAX-WS presenta las siguientes características:

- proporciona una serie de anotaciones que facilitan la tarea a los desarrolladores.

- Permite implementar e invocar operaciones de servicios web basados en el protocolo SOAP.

- Permite operaciones asíncronas.

- Proporciona mapeado WSDL para Java, mediante un compilador de Java a WSDL, que genera el documento WSDL correspondiente a cualquier clase o interfaz Java.

- El mapeado WSDL se realiza mediante JAXB. JAXB es un compilador que construye un árbol de objetos Java representado en un documento XML, permitiendo de esta manera manipular a los desarrolladores dichos objetos fácilmente.

8.1.4. Google Maps API

Las tecnologías de Google suelen ofrecer APIs bastante completas. Dentro de todas estas, hemos usado la API escrita en JavaScript de Google Maps.
Esta tecnología nos ofrece posicionamiento según puntero, según coordenadas, según nombre de localización, y geolocalización para saber la posición donde nos encontramos. Ofrece también cálculo de distancias según el medio de transporte y la hora. Podemos ir en coche, transporte público, andando, barco, ...
Los servicios que ofrece esta API son gratuitos, aunque con ciertas restricciones como la cuota de uso. Sin embargo, para los requerimientos de nuestro proyecto esto no fue un problema.

Estos servicios nos fueron útiles para ejecutar los algoritmos implementados para resolver el problema del TSP sobre puntos marcados en un mapa, para posteriormente dibujar la ruta de la solución obtenida en él.

8.2. Arquitectura cliente-servidor

Debido a que hemos implementado un cliente web en el que ejecutar y mostrar los resultados de los algoritmos implementados para cada problema, hemos establecido una arquitectura cliente-servidor. De esta forma pudimos crear dos estructuras independientes conectadas para un mismo fin.
Como alternativa podríamos haber tenido todo en el mismo cliente web, y utilizar lenguajes como JavaScript o PHP para implementar los problemas y los algoritmos, o usar PHP para servidor y cliente.

8.2.1. Servidor

8.2.1.1. Organización y Estructura

Al no saber bien en el comienzo cómo hacer el proyecto, utilizamos una arquitectura multicapa dividiendo la aplicación en tres partes: Vista, Negocio e Integración, usando el patrón de diseño MVC. En un inicio, modelamos de esta forma el proyecto porque supusimos que lo íbamos a necesitar.
Después pasamos a arquitectura cliente-servidor por lo que la capa de vista desapareció, y conservamos la capa de integración porque pensamos que podría ser útil tener algún banco de trabajo de gran volumen, y que este se pudiera almacenar en una base de datos. También pensamos almacenar dentro de la base de datos alguna estructura de los problemas a resolver, tales como un mapa o una mochila, que en una base de datos relacional podrían resultar fáciles de estructurar. Un mapa consistiría ciudades asociadas con una tabla de distancias y para una mochila, se podría tener una tabla de ítems con atributos tales como id, coste y beneficio. Pero debido a que no importa el tamaño y que esto supondría complicar la aplicación, descartamos definitivamente esta capa también.

Al final nos ha quedado dentro de servidor una capa de negocio puramente.

Esta capa de negocio, la hemos dividido en tres paquetes importantes, uno dedicado a los problemas, otro a algoritmos y otro más para pruebas. Además, existen otros paquetes para manejo de ficheros, *parsers* o construcción de mapas.

Dentro del paquete dedicado a los algoritmos tenemos unos esquemas básicos que deben seguir las implementaciones de los mismos. Además, puesto que hemos hecho un *framework* genérico para los algoritmos genéticos, también están dentro todas las estructuras necesarias, sus partes e interfaces.

Dentro del paquete para los problemas *(TSP y Mochila 0-1), tenemos* los paquetes necesarios para la implementación de los mismos, es decir las *estructuras* necesarias para

modelizarlos y resolverlos. Estas estructuras pueden ser los *mapas* o las *mochilas* e *ítems*, además de la *representación de las soluciones.*

Los mapas fueron sacados fuera durante un tiempo porque para algunas metaheurísticas como en el caso de ACO, pensamos que se podría adaptar su funcionamiento probado en el TSP sobre el problema de la Mochila 0-1, transformando de alguna manera la estructura mochila a mapa, por lo que debía estar fuera, pero como encontramos otra manera de adaptarlo, la cual explicaremos más adelante, mapa volvió a pertenecer a TSP.

8.2.1.2. Patrones de diseño

En cuanto a los patrones de diseño utilizados dentro de la parte servidor escrita en Java, tenemos:

- *Strategy*: separa la implementación de la estructura. A través de una clase específica, este patrón permite implementar distintos comportamientos encapsulados en clases hijas. Este patrón lo utilizamos en todos los algoritmos, puesto que tenemos una interfaz que muestra el comportamiento, las operaciones... que deben tener todos los algoritmos.

- *Application Service*: Este patrón se encuentra en el límite entre la capa de presentación y la de negocio. Engloba y agrega componentes de esta segunda capa, proporcionando una capa uniforme de servicios listos para ser utilizados. Este patrón es utilizado para implementar los servicios web, que se reducen a resolver un determinado problema mediante un algoritmo dado.

- *Factorías*: Separan la construcción del objeto de su clase, de manera que para construir ciertos objetos no es necesario saber su representación. Es usado en todas las partes importantes de la aplicación, para construir algoritmos, mapas, mochilas, etc.

- *Builder*: Sirve para construir objetos complejos, es decir, formados por partes que no se indican en tiempo de compilación, debido a que existen diversas configuraciones para crear dichos objetos. Se utiliza en la creación de los Algoritmos Genéticos.

- *Transfer*: encapsula un conjunto de datos destinado a transmitirse de una capa a otra. Es utilizado por ejemplo al pasar la solución a una instancia de un problema desde el servidor al cliente.

- *Adapter*: Cuando desde una clase se quiere utilizar otra ya implementada, pero cuya interfaz no es la adecuada para lo que se necesita, este patrón permite que

ambas clases puedan trabajar juntas, aunque sus interfaces incompatibles. Nos sirvió para utilizar objetos como si fueran otros, añadiendo métodos o atributos. Lo utilizamos para transformar ciertos algoritmos en otros, por ejemplo, para adaptar los Algoritmos Genético a la resolución del TSP.

8.2.1.3. Diagramas de clases

Debido a que los diagramas de clases pueden llegar a ser voluminosos, pedimos al lector que se dirija a los *apéndices* de esta memoria, los cuales están dedicados íntegramente a esta sección. Los diagramas que se presentan en estos apéndices están dedicados a las partes que hemos considerado importantes en la implementación del lado del servidor: algoritmos, mapas, mochilas, servicios web y framework para algoritmos genéticos.

8.2.2. Cliente

Cuando nos pusimos a pensar en cómo y con qué herramientas desarrollar nuestro cliente web, como primera opción surgió la de utilizar principalmente tecnologías de Java EE (Java Enterprise Edition). Concretamente teníamos pensado desarrollar la lógica de cada una de nuestras páginas mediante JSP (Java Server Pages) y sus anotaciones. Por otro lado, para realizar la interacción entre el cliente y el servidor, por medio del patrón de diseño MVC, pensamos como
Sin embargo, debido a que nos enteramos de que el explorador web de Google (Google Chrome) dejaría de soportar ciertas herramientas basadas en java[16] como los Applets o ciertos plugins, decidimos no arriesgar y descartar la opción de las tecnologías java como herramientas principales para el desarrollo del cliente web. Como queríamos que nuestro trabajo pudiese llegar alguna vez a ser útil para otros, surgió la idea de crear y proporcionar servicios web que resuelvan problemas mediante distintos tipos de algoritmos. Estos servicios web fueron consumidos desde el cliente web.

Así que al final, utilizamos únicamente tecnologías Java en el lado del servidor y para el lado del cliente utilizamos las tecnologías web ya presentadas anteriormente (HTML5, CSS3, JavaScript, JQuery, AJAX, JSON, PHP).

En lo que respecta a esta sección, presentaremos y explicaremos el funcionamiento de las interfaces web del cliente, con las que el usuario puede interactuar y realizar pruebas de resolución de distintas configuraciones de los problemas del TSP y de la Mochila 0-1.

8.2.2.1. Páginas web

Al igual que ocurre con los servicios web, la aplicación cliente está alojada en nuestro servidor local. Una vez que se accede, la página principal es meramente informativa. La página es la siguiente:

Figura 8.3. Página principal de la aplicación web cliente

A partir de esta página, se puede acceder a las páginas dedicadas a la resolución de cada uno de los problemas utilizando los enlaces de la cabecera de la página:

Figura 8.4. Barra de navegación de la aplicación web cliente

Esta cabecera está presente en todas las páginas, haciendo la navegación más cómoda al usuario.

A continuación, se presentan las interfaces de las dos páginas dedicadas a la resolución de los problemas del TSP y la Mochila 0-1 respectivamente:

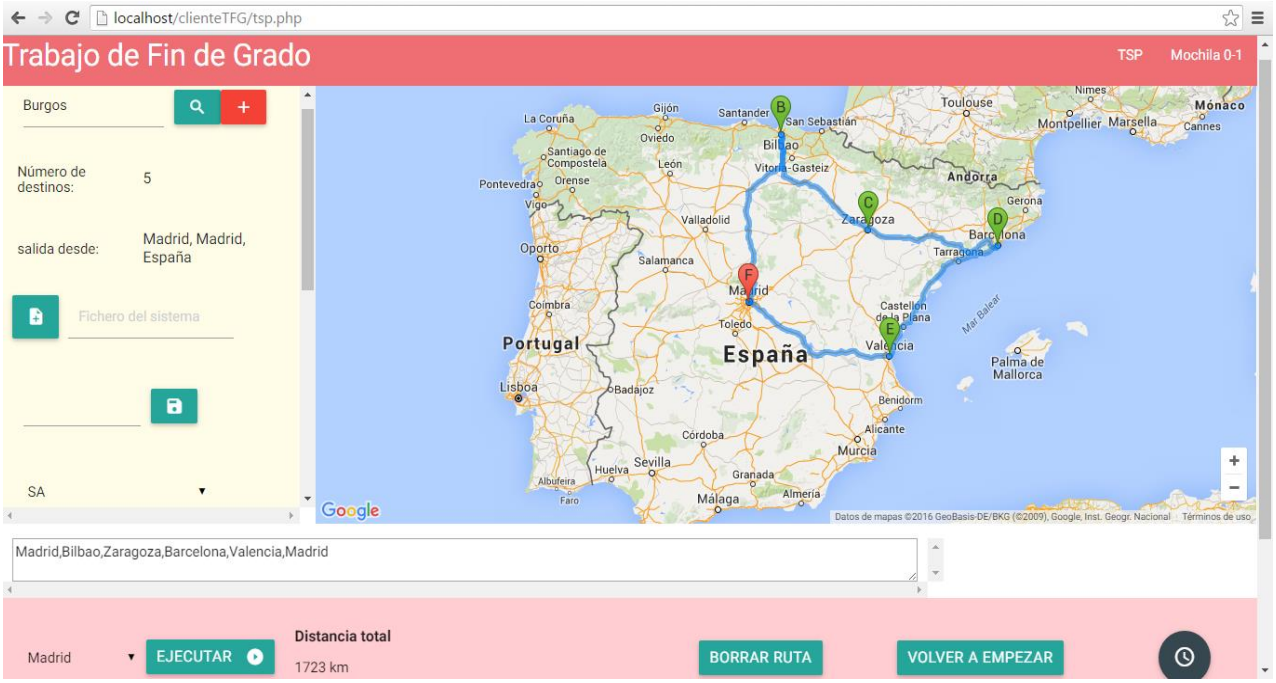


Figura 8.5. Página del resolutor del TSP de la aplicación web cliente

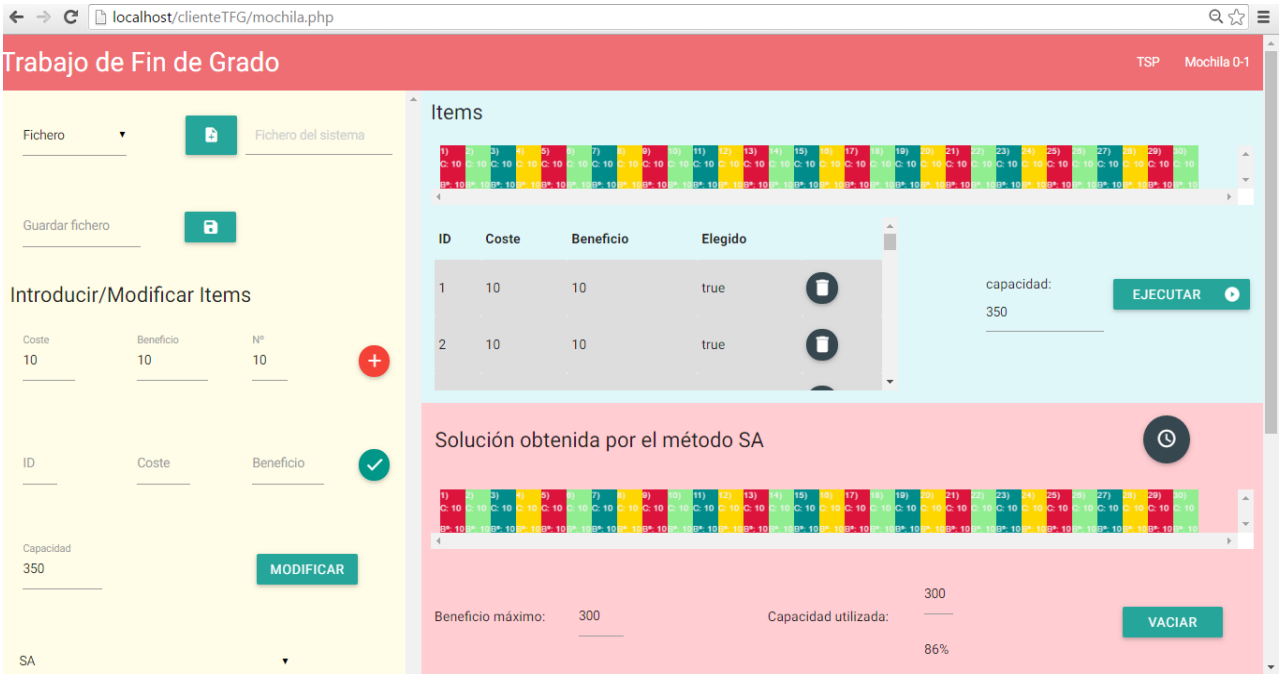


Figura 8.6. Página del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

8.2.2.1.1. *Resolutor* TSP

La página para la resolución del TSP, está dividida en tres secciones, cada una con una función distinta:

1. *Sección de configuración del problema*:

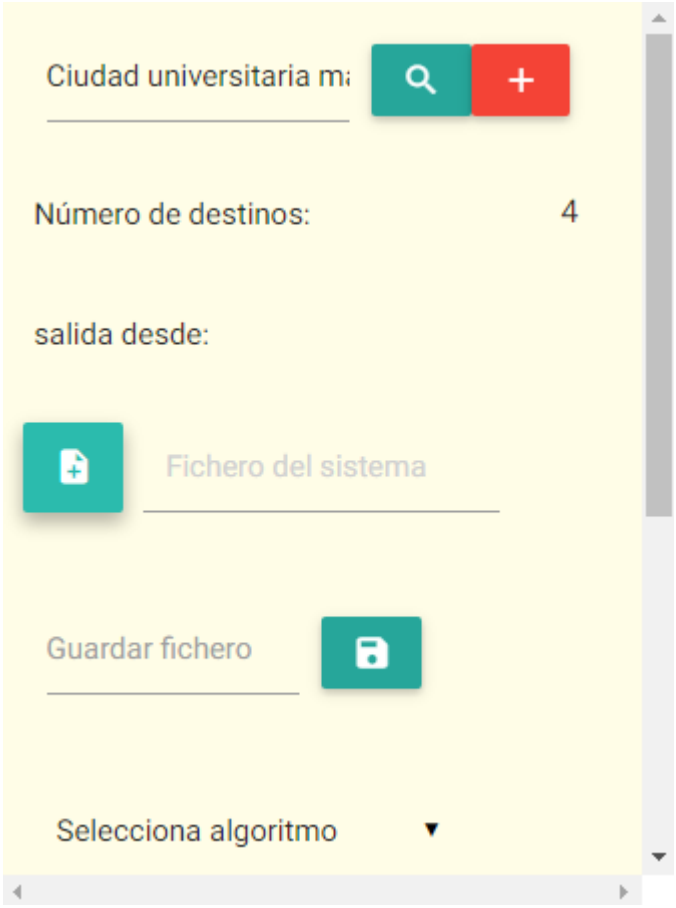


Figura 8.7. Sección de configuración para el resolutor del TSP de la aplicación web cliente

Se dispone de un buscador, donde puede introducir el nombre del punto que desee marcar en el mapa. En caso de existir, se le ofrecerá al usuario la oportunidad de poner una marca sobre dicho punto en el mapa.

Figura 8.8. Ejemplo de búsqueda y marcado de ciudades para el resolutor del TSP de la aplicación web cliente

También se pueden marcar puntos en el mapa por medio de ficheros de texto que almacenan los nombres de dichos puntos. Para ello se le ofrece al usuario un selector de archivos a través del cual puede seleccionar un fichero de su sistema.

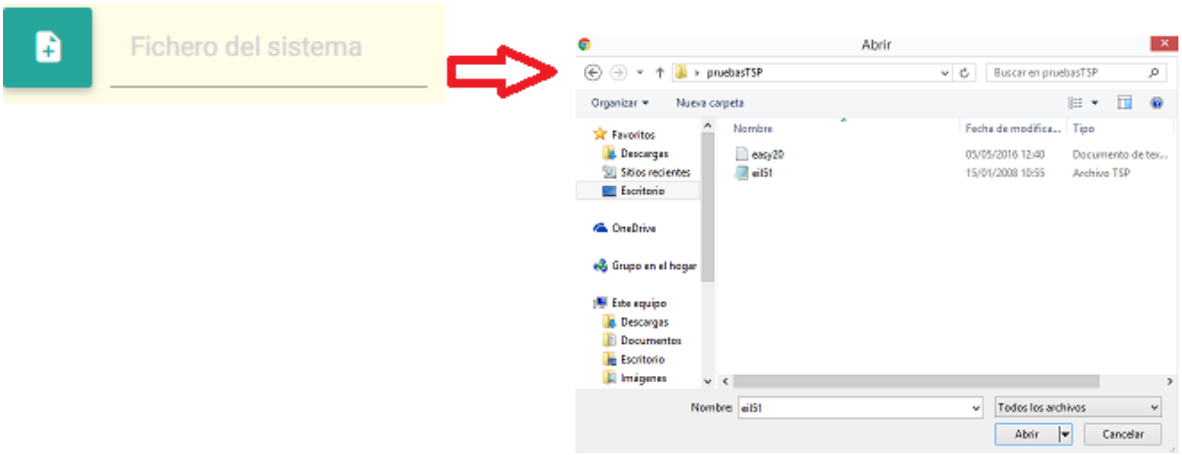


Figura 8.9. Ejemplo de carga de ciudades mediante fichero de texto para el resolutor del TSP de la aplicación web cliente

Por otro lado, como es lógico, se le ofrece al usuario la opción de seleccionar el tipo de algoritmo con el que quiere resolver el problema por medio de una lista desplegable. En el caso de los algoritmos heurísticos, estos pueden tener parámetros de configuración, por los que una vez seleccionados, se muestran dichos parámetros para que el usuario los pueda ajustar a su gusto.

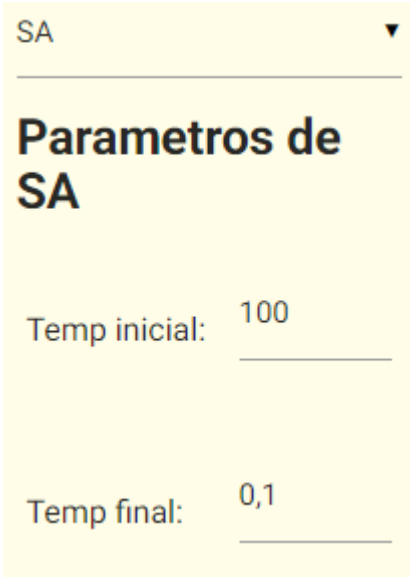


Figura 8.10. Ejemplo de selección y configuración de algoritmos para el resolutor del TSP de la aplicación web cliente

Además de todo esto, esta sección de la página muestra información al usuario

sobre el número total de puntos marcados en el mapa y el punto de partida desde el cual resolver el problema.

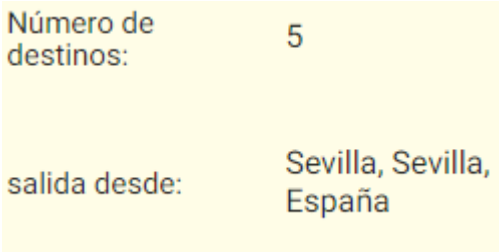


Figura 8.11. Información sobre el número de ciudades seleccionadas y el punto de partida para el resolutor del TSP de la aplicación web cliente

2. *Sección del mapa*:

Figura 8.12. Mapa de Google Maps del resolutor del TSP de la aplicación web cliente

Esta sección está compuesta únicamente por un mapa del mundo que proporciona la API de Google Maps. Sobre este mapa se dibujan los marcadores en los puntos de destino buscados y añadidos por el usuario, pudiendo también eliminarlos haciendo *doble click* sobre ellos. También se dibuja la ruta de la solución obtenida por el algoritmo utilizado.

Figura 8.13. Ejemplo de marcado de ciudades y trazado de rutas sobre el mapa de Google Maps del resolutor del TSP de la aplicación web cliente

3. *Sección de ejecución y resultados:*

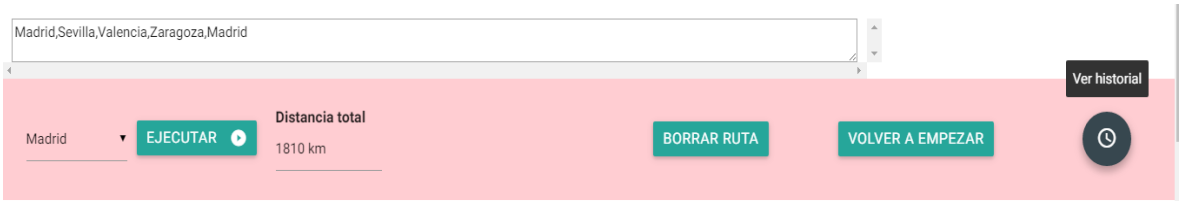


Figura 8.14. Sección de ejecución y muestra de resultados del resolutor del TSP de la aplicación web cliente

Esta última sección está compuesta por los siguientes elementos:

- Una lista desplegable para que el usuario pueda seleccionar el punto de partida de la ruta.

- Un botón para empezar la ejecución de la resolución del problema.

- Texto informativo sobre la distancia total en km de la ruta solución obtenida.

- Un botón para borrar la ruta dibujada en el mapa, de modo que se vuelvan a mostrar los marcadores.

- Un botón para borrar el contenido del mapa, que borrará tanto los marcadores como la ruta dibujada.

- Un botón para visualizar el historial de problemas resueltos.

Este historial de resultados tiene como columnas de información: el número de ciudades del problema, el método utilizado en la resolución, los parámetros de dicho método y la distancia total de la ruta solución obtenida.

Historial de resultados

Nº ciudades	Método	Parámetros	Distancia	Ruta
5	SA	Ti:100,Tf:0.1,km:1000,c:100 iteraciones: 100	2574 km	Valencia,Barcelona,Galicia,Madrid,Alicante,Valencia
5	GA	Tam poblacion :1000,Mutacion :0.001,Cruce :0.7,Elitismo :true,Num generaciones :1, Memetismo:true iteraciones: 100	2584 km	Madrid,Galicia,Barcelona,Valencia,Alicante,Madrid
7	ACE	IterMax:5000,NumHormigas:2 iteraciones: 500000	3073 km	Sevilla,Badajoz,Galicia,Madrid,Barcelona,Valencia,Alicante,Sevilla

Figura 8.15. Historial de resultados del resolutor del TSP de la aplicación web cliente

8.2.2.1.2. *Resolutor* Mochila 0-1

Al igual que con la página del *resolutor* del TSP, podemos dividir también la página del *resolutor* del problema de la Mochila 0-1 en tres secciones:

1. *Sección de configuración del problema*:

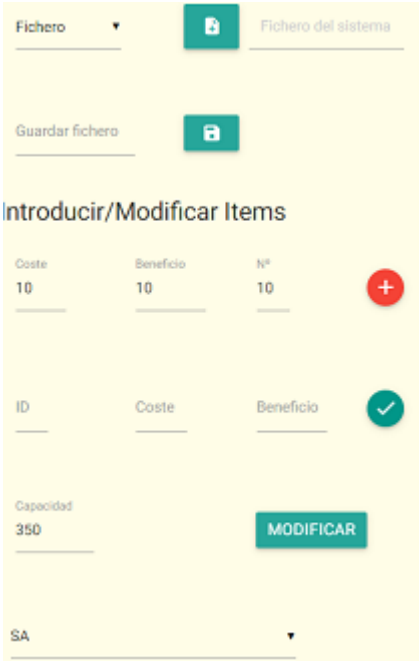


Figura 8.16. Sección de configuración para el resolutor del problema de la Mochila 0-1 de la aplicación web cliente

En esta sección el usuario podrá introducir ítems al problema de forma manual, rellenando los campos de coste, beneficio, Nº de ítems a introducir. El campo Nº de ítems permite introducir más de un ítem con el mismo coste y beneficio. En el siguiente ejemplo se añaden 10 ítems, cada uno con un coste de 10 y un beneficio de 10.

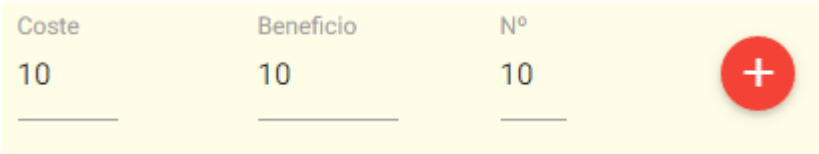


Figura 8.17. Ejemplo de introducción de ítems del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Al igual que en el *resolutor* del TSP, También se dispone de un selector de archivos para que el usuario pueda introducir ítems al problema por medio de algún fichero de texto de su sistema.

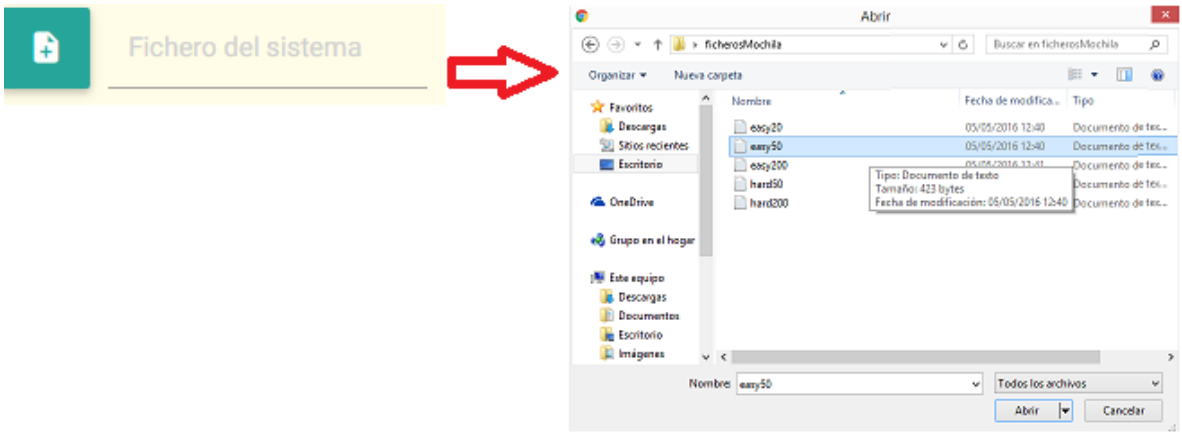


Figura 8.18. Ejemplo de carga de ítems mediante fichero de texto para el resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Los ítems introducidos pueden ser modificados, en cuanto a su coste y/o beneficio. Para ello el usuario ha de introducir el ID del ítem que desea modificar y los nuevos valores del coste y el beneficio de dicho ítem.

Figura 8.19. Ejemplo de modificación de ítem del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Así mismo, la capacidad de la mochila del problema puede ser modificada.

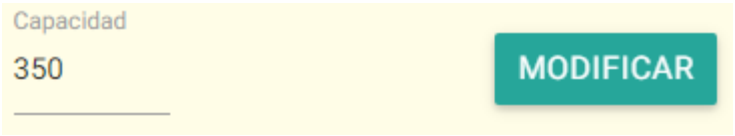


Figura 8.20. Ejemplo de modificación de la capacidad de la mochila del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Como en el *resolutor* del TSP, también se le ofrece al usuario la opción de seleccionar el tipo de algoritmo con el que quiere resolver el problema por medio de una lista desplegable, y la posibilidad de configurar a su gusto los parámetros del algoritmo seleccionado.

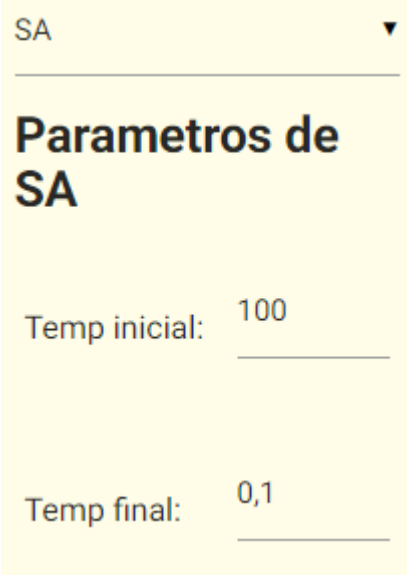


Figura 8.21. Ejemplo de selección y configuración de algoritmos para el resolutor del problema de la Mochila 0-1 de la aplicación web cliente

2. *Sección de la mochila y conjunto de ítems*:

Figura 8.22. Sección de la mochila, conjunto de ítems y ejecución del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Esta sección contiene los elementos de los que está compuesto el problema: la mochila y su capacidad, y el conjunto de ítem:

El conjunto de ítems está diseñado mediante elementos canvas de HTML5. Cada ítem del conjunto está dibujado en forma de rectángulo.

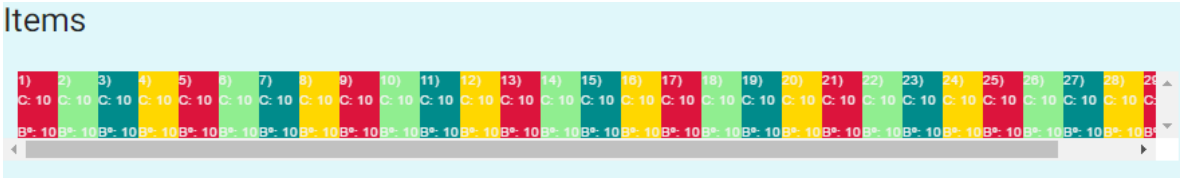


Figura 8.23. Conjunto de ítems del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Para que el usuario pueda verificar la información de cada ítem del conjunto, dispone de una tabla donde por cada fila se muestra el ID, coste, beneficio y un valor booleano para informar sobre si el ítem en cuestión está seleccionado o no como parte de la solución del problema. Además, cada ítem tiene asociado un botón para eliminarlo si así se desea.

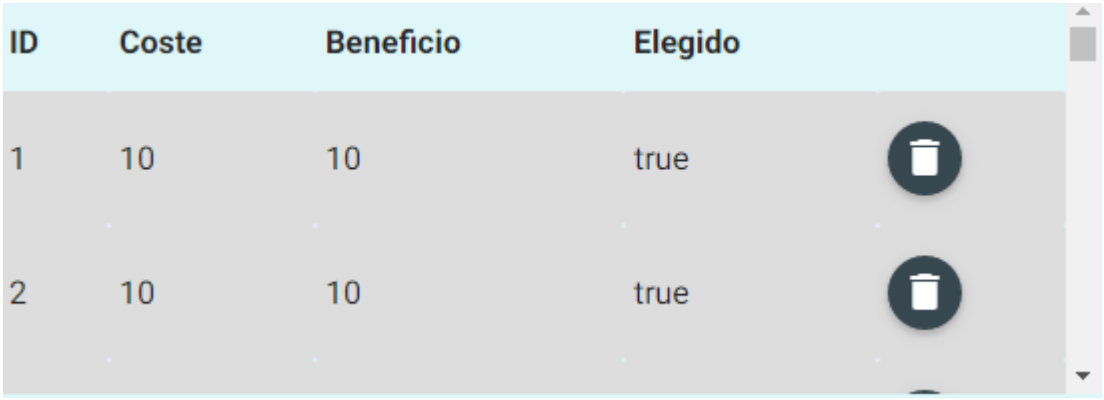


Figura 8.24. Tabla de información de los ítems del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

También se dispone de un botón para empezar la ejecución de la resolución.

3. *Sección de resultados*:

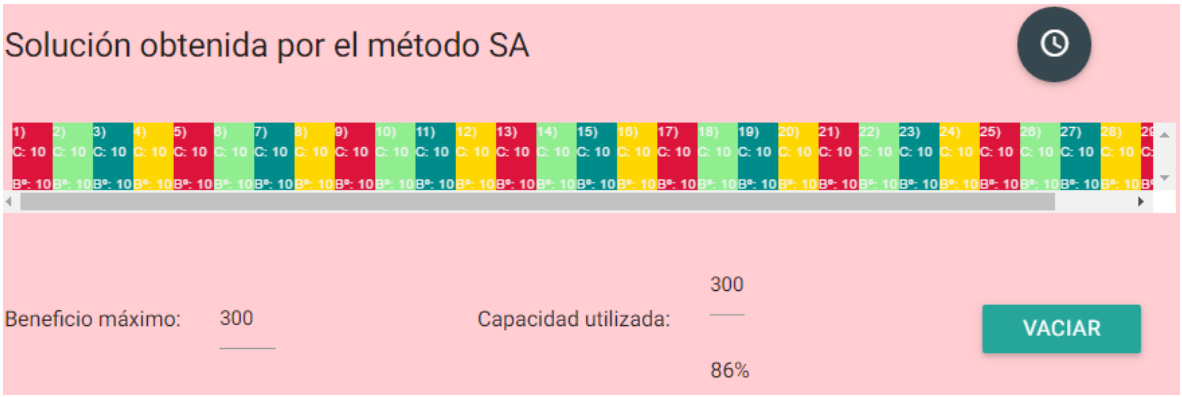


Figura 8.25. Sección de muestra de resultados del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

La sección de resultados está compuesta por el subconjunto de ítems seleccionados para ser introducidos en la mochila del problema. Cada ítem de dicho subconjunto está también

representado por un elemento *canvas* de HTML5.

Debajo del subconjunto de ítems seleccionados, se muestra más información sobre la solución obtenida. Se muestra el beneficio máximo obtenido y la capacidad total ocupada en la mochila.

También se dispone de un botón para vaciar el conjunto de ítems y la mochila, de manera que el usuario pueda volver a empezar a configurar una nueva instancia del problema.

Por último, se dispone de un botón para visualizar el historial de problemas resueltos.

Metodo	Parametros	Beneficio	Coste	Capacidad	Num. elegidos	Num. items
ACE	IterMax:5000,NumHormigas:2 iteraciones: 500000	296	311	323	28	30
SA	Ti:100,Tf:0.1,km:1000,c:1 iteraciones: 100	296	311	323	28	30
Grasp	h:0.78,i:0.01 iteraciones: 1	296	311	323	28	30
Grasp	h:0.78,i:3.2 iteraciones: 320	296	311	323	28	30

Figura 8.26. Historial de resultados del resolutor del problema de la Mochila 0-1 de la aplicación web cliente

Capítulo 9

Conclusiones y trabajo futuro

9.1. Conclusiones

Al finalizar este trabajo, ha llegado el momento de mirar hacia atrás para hacer un recuento de lo que hemos conseguido:

- Hemos conseguido estudiar y comprender los dos problemas (TSP y el problema de la Mochila 0-1) en los que nos hemos centrado. Hemos sido capaces de establecer un modelo matemático con el que trabajar, y que nos ha ayudado en las labores de crear las estructuras de datos necesarias para los problemas y de asegurar su buen funcionamiento.

- Se han estudiado y comprendido diversos tipos de heurísticas y metaheurísticas, y se han aplicado a la resolución de los problemas mencionados. Esto nos ha aportado los conocimientos suficientes para sentar una base a partir de la cual poder seguir adquiriendo más conocimientos de este bonito campo de investigación.

- Hemos sido capaces de trabajar en nuestra capacidad de interpretación y análisis de resultados cuando nos ha tocado realizar un estudio comparativo entre las heurísticas, metaheurísticas y métodos exactos aplicados a la resolución de los problemas.

- Hemos conseguido desarrollar una aplicación software con la que realizar una presentación más amigable de la resolución de los problemas. Durante esta etapa, hemos tenido que enfrentarnos a nuevas tecnologías, las cuales hemos aprendido a utilizar. Por otro lado, creemos que hemos conseguido diseñar una buena arquitectura software sobre la que hemos desarrollado nuestro sistema y que podría servirnos para ampliar este trabajo o para empezar el desarrollo de un nuevo proyecto.

- También ha sido importante la labor de organización del trabajo, en cuanto a objetivos, reparto de tareas, tiempos que cumplir, etc.

Ahora bien, y como es lógico, en todo proyecto siempre surgen problemas y nuestro caso no ha sido una excepción. Al iniciar este Trabajo de Fin de Grado, el campo de los métodos metaheurísticos era desconocido para nosotros, pero poco a poco fuimos adquiriendo mayor habilidad en cuanto a su estudio e implementación, por lo que podemos decir con toda seguridad que esta experiencia ha sido muy enriquecedora. Tras haber finalizado nuestra etapa de estudiantes y viendo ésta con retrospectiva, nos hubiese gustado poder haber tenido contacto con este tema en

alguna asignatura de nuestro plan de estudios, y desde nuestra parte, animamos a que se pueda hacer posible en algún momento.

En lo que concierne al estudio de los problemas, al comienzo teníamos el deseo de poder estudiar el mayor número posible de problemas NP, sin embargo, la falta de experiencia y el hecho de tener que realizar un estudio previo para poder comprenderlos, así como también tener que seleccionar las metaheurísticas y estudiar su adaptación a dichos problemas, supusieron replantearnos en cierta medida este objetivo, por lo que decidimos seleccionar dos problemas bien diferenciados entre sí.

En cuanto al desarrollo de la aplicación, en la parte del servidor introdujimos toda la lógica de las metaheurísticas, de los problemas y la implementación de los servicios web, mientras que en la parte del cliente web se implementaron los *resolutores* de los problemas. Estas partes también sufrieron cambios a lo largo de este trabajo, principalmente porque en un inicio no teníamos la idea de desarrollar servicios web para la resolución de los problemas, pero pensamos que dichos servicios web podrían llegar en algún momento a ser útiles.

A pesar de todos los cambios en nuestros planes y de los problemas que tuvimos a lo largo del camino, creemos que hemos cumplido con los objetivos presentados en el capítulo 1, y finalmente, estamos satisfechos del trabajo realizado.

9.2. Trabajo futuro

Debido al tiempo que requiere el estudio de metaheurísticas, y su adaptación a la resolución de los problemas que se nos plantean, no hemos podido estudiar tantas como nos hubiese gustado. En un futuro nos gustaría poder estudiar otros tipos de metaheurísticas aplicadas a otros problemas que resulten también interesantes. Creemos que el diseño de nuestro sistema actual podría facilitar dicha ampliación de este estudio.

Por otro lado, y aunque no sea el tema principal de este trabajo, creemos que podríamos realizar ciertas mejoras en nuestros servicios web para la resolución de problemas. Entre estas mejoras, nos gustaría realizar una mejor definición de los mismos, es decir, una mejor definición de sus parámetros de entrada y salida, con el fin de que resulten más útiles para otras personas que deseen realizar pruebas.

En cuanto a nuestros *resolutores* de problemas en nuestro cliente web, sobre todo, nos gustaría explotar más la API de Google Maps para mejorar el *resolutor* del TSP. Nos gustaría, por ejemplo, poder obtener rutas para otros medios de transporte, ya que actualmente solo obtenemos rutas para coche.

Por último, nos hubiese gustado darle un uso más real a las metaheurísticas estudiadas, como, por ejemplo, para intentar optimizar redes de transporte en una ciudad, que fue la primera idea de nuestro tutor, pero que no pudo realizarse debido a la dificultad de obtención de datos sobre la intensidad de tráfico.

Chapter 10

Conclusions and future work

10.1. Conclusions

Once this work is completed, it is time to look back and summarize what we have achieved:

- We have studied and understood the two problems (TSP and 0-1 Knapsack problem) in which we focused. We have been able to establish a mathematical model to work with. It helped us in the task of creating the necessary data structures for problems and to ensure their right operation.

- We have studied and understood various types of heuristics and metaheuristics, and they have been applied to solve these problems. This study has established a suitable base to further expand our knowledge in this beautiful field of research.

- We have been able to interpret and analyze the results obtained to make a comparative study among the heuristics, metaheuristics, and exhaustive methods applied for solving problems.

- We have developed a software application to perform a more user-friendly presentation of the resolution of the problems. During this stage, we had to deal with new technologies, which we learned to use. On the other hand, we believe we have designed a good software architecture and as a result it would be simple to extend this work or to start developing a new project.

- It has also been important the scheduling work, in terms of objectives, distribution of tasks, timeouts, ...

However, and not surprisingly, in every project always arise problems and our case has not been an exception. When we started this final degree project, the field of metaheuristic methods was unknown to us, but we gradually gained skills in their study and implementation, so we can ensure that this experience has been very enriching. After finishing our students stage, we would have liked to have studied this theme in a subject of our faculty syllabus, and we encourage teachers to make it possible.

Regarding the study of the problems, initially we had the desire to study as many NP-problems as possible, however, the lack of experience, the previous study we did in order to

understand them, the selection of metaheuristics, and the study of their adaptation to these problems, entailed us to rethink this goal, so we decided to select two very different problems.

Concerning the development of the application, on the server side we introduced all the logic of metaheuristics, the problems and the implementation of web services, while on the web client side the solvers of the problems were implemented. These parts were changed throughout this work, mainly because initially we had no idea of developing web services for the resolution of problems, but we think these could be useful.

Despite all the changes in our plans and the problems we had along the way, we believe we have met the objectives presented in Chapter 1, and finally, we are satisfied with the work done.

10.2. Future work

Because of the time required for the study of metaheuristics and their adaptation to the resolution of the problems, we have not been able to study as many as we would have liked. In the future we would like to study other types of metaheuristics applied to other interesting problems. We believe he design of our current system could facilitate the extension of this study.

Moreover, we believe we could make certain improvements to our web services for problem resolution. One of these improvements would be to make a better definition of them, that is we could improve their input and output parameters, in order to make them more useful to others.

As for our problem solvers in the web client, we would like to exploit more the Google Maps API to improve the TSP solver. For example, we would like to obtain routes to other transport modes, since currently we only get car routes.

Finally, we wished to give a more realistic use metaheuristics studied, for example, to try to optimize transport networks in city, that it was the first idea of our advisor, but it could not be performed due to the difficulty of obtaining traffic intensity data.

Capítulo 11

Aportaciones individuales

11.1. Por parte del alumno Jhonny Vargas Paredes

Mis aportaciones en este trabajo han ido variando a lo largo de este y he participado en buena parte del mismo.

En cuanto a los problemas NP a estudiar, en un comienzo empecé por investigar sobre ellos para seleccionar los más adecuados y diferenciados entre sí con los que realizar este estudio. Durante esta etapa tuve que descartar varios de ellos y centrarme sobre todo en el TSP, el cual fue nuestro primer problema base a abordar, y sobre el cual probar las implementaciones de las metaheurísticas implementadas. Además, para este problema, investigué métodos de optimización tales como k-opt, que nos sirvieron para mejorar más aún si cabe las metaheurísticas.

El siguiente problema elegido fue el problema de la Mochila 0-1, en el cual realicé algunas tareas, sobre todo de revisión de la implementación, para asegurar la fiabilidad de las estructuras de datos para la modelización del problema, con el fin de facilitar todo lo posible la adaptación de los algoritmos implementados para su resolución.

A lo largo de toda esta etapa de estudio de los problemas, tuve que descartar varios de ellos sobre los que había estado investigando. Entre estos problemas, estaban el SAT (problema de satisfacibilidad boolena) y *Job shop scheduling*, los cuales no llegaron a ser implementados por falta de tiempo.

En la parte de las metaheurísticas, mi compañero y yo repartimos el trabajo, por lo que sobre todo me centré en el estudio e implementación de las metaheurísticas GRASP (Mochila 0-1) y Simulated annealing. No obstante, ambos siempre estuvimos en contacto para ayudarnos si era necesario, de manera que, por ejemplo, ayudé en el estudio de la metaheurística ACO y en la optimización de otras metaheurísticas. Esto ayudó a que ambos pudiésemos aprender los conceptos de cada una de las metaheurísticas estudiadas.

En cuanto a los métodos exactos, realicé tareas de estudio e implementación de algoritmos de programación dinámica.

He realizado labores de representación, análisis e interpretación de resultados, para poder realizar las comparativas de los métodos algorítmicos implementados.

En la parte del diseño del sistema, sobre todo realicé tareas para la integración del cliente web con el servidor. En un comienzo realicé un estudio de las tecnologías a utilizar (*Java Struts* entre ellas) y finalmente surgió la idea de los servicios web como herramienta para el desarrollo del cliente. Por lo que investigué sobre las tecnologías a utilizar para el desarrollo de estos servicios web en el servidor, centrándome en los protocolos y en las tecnologías Java que facilitasen su implementación (haciendo énfasis en JAX-WS).

Por otro lado, en la parte del cliente, en un principio pensamos en utilizar PHP como lenguaje de programación para consumir los servicios web del servidor, por lo que preparé el cliente web para ello. Sin embargo, después decidimos utilizar JavaScript, por lo que realicé una tarea de investigación de las tecnologías a utilizar para consumir los servicios web con este lenguaje, el cual está orientado al lado del cliente. Sobre todo, me centré en el estudio de AJAX. Finalmente, implementé el sistema necesario para ello.
También realicé tareas de organización del código que forma parte del cliente web, creando partes reutilizables, organizando librerías y dependencias, etc.

En cuanto a los *resolutores* de los problemas implementados en el cliente, desarrollé la interfaz para el *resolutor* del TSP y colaboré con mi compañero para la utilización de la API de Google Maps. Para esto tuve que estudiar los servicios ofrecidos por esta API escrita en JavaScript, para utilizarlos en todas aquellas partes necesarias, como son la creación de mapas o el trazado de rutas, entre otros.

En la parte del servidor, implementé los servicios web destinados a la resolución del TSP y que sirvieron de base para implementar lo servicios web para la resolución del problema de la Mochila 0-1.

La mejora visual y la parte de interacción amigable de las interfaces web del cliente con el usuario, fue un tema que llevé a cabo prácticamente en su totalidad, utilizando Materialize, pudiendo mi compañero participar, por ejemplo, en la mejora de la interfaz del problema de la Mochila 0-1. Creo que en la medida de lo posible el trabajo, proporcionó buenos resultados.

Mi labor también estuvo dedicada a la revisión de prácticamente todas las partes de este trabajo y a lo largo de todo el proceso de desarrollo. Realicé esta labor en las estructuras de datos para los problemas, mis metaheurísticas, los servicios web, interfaces de usuario en el cliente web, comunicación cliente-servidor, etc.

Implanté un repositorio de código para el desarrollo del cliente web, por lo que tuve que aprender a desenvolverme con este modo de trabajo, produciéndose a veces problemas que pude solventar.

La planificación del trabajo y el tiempo, fue un tema que realizamos entre mi compañero y yo. Para ello se repartieron los esfuerzos y tiempos de modo que pudiésemos cumplir con nuestros objetivos, teniendo siempre en cuenta nuestros otros compromisos académicos y laborales.

También, he realizado una labor intensa e importante en el desarrollo de esta memoria, tanto en organización, contenido, y sobre todo en redacción y corrección.

Por último y como opinión personal, quiero agradecer a mi compañero por su trabajo realizado. Las aportaciones de ambos ya sean pocas o muchas han sido siempre importantes para poder sacar adelante este trabajo.

11.2. Por parte del alumno Víctor Penit Granado

Mi aportación en el proyecto ha sido muy intensa y variada durante el proyecto de manera que he participado en gran parte de la realización de este. Mi labor se ha enfocado en todas las áreas de comprenden este trabajo.

La parte de investigación, la cual he hecho desde el principio, empezó con una pesquisa inicial sobre los problemas de Karp y las metaheurísticas posibles. Fruto de esto fue encontrar la clasificación de los problemas e investigar a fondo sobre el TSP, problema que nuestro tutor pidió como inicial. Además, investigué otros problemas por encima, como, por ejemplo, satisfacción booleana y asignación de tareas, y una serie de metaheurísticas variadas, entre las cuales se encuentran, los algoritmos evolutivos (genéticos), algoritmos inspirados en las colonias de hormigas como ACO, de los que ya tenía conocimiento, Simulated Annealing, PSO (Particle Swarm Optimization), que seguí investigando más adelante, además de otras que fueron estudiadas pero abandonadas, como por ejemplo, búsqueda tabú. El objetivo de mi investigación era una ambiciosa meta de estudiar tantas metaheurísticas como pudiera, pero se estableció como requisito que elegiría solo 2 de esas, y por su implementación relacionada con la naturaleza, me decanté por ACO y algoritmos genéticos. Por lo que, al cabo del tiempo, a pesar de haber empezado una implementación bastante básica de PSO, fue abandonada y más tarde, por razones variadas, fue sustituida por GRASP. Destacar que la mayor parte de la investigación fue en torno a los algoritmos genéticos, teniendo como resultado, un framework bastante completo.

Al igual que la tarea anterior, desde el inicio realicé una labor de diseño, para crear un buen sistema flexible, dividido en paquetes, con los patrones adecuados para fomentar las buenas características de la orientación a objetos. Como es normal el área de implementación es la que más tiempo y esfuerzo me ha ocupado. Empezando por la implementación del sistema, las interfaces correspondientes, en cada problema las estructuras necesarias para encapsular y resolver los problemas, como, por ejemplo, los mapas, las mochilas, las soluciones de estos, las cuales pueden ser transferidas entre las capas, soluciones que encapsulan lo necesario para poder interpretar los resultados de las pruebas, ... Una vez establecido un sistema básico inicial, pude empezar a implementar algoritmos simples como Random y BackTracking, los cuales tenían dos objetivos: poder realizar pruebas para verificar el sistema y así arreglar los posibles errores antes de incluir algoritmos más complejos y disponer de unas heurísticas con las que comparar las metaheurísticas e incluso, crear soluciones iniciales. Dentro de estos métodos podemos encontrar por ejemplo el vecino más cercano, para el cual creé el 2-opt y lo usé más tarde para optimizar la mayoría de los algoritmos; y el método de inserción en el TSP o la selección ordenada por el ratio beneficio / coste en el Problema de la Mochila 0-1.
Dentro de las metaheurísticas, también he implementado y adaptado entre otros el ACO y debido a lo que investigué, pude ver las diferencias en la implementación de ACO normal y ACO elitista. Otra aportación ha sido colaborar en Simulated Annealing, al investigar e iniciar la implementación, aunque la versión final fue la de mi compañero, que la volvió a implementar desde cero, y la realización de todo el algoritmo GRASP del TSP.

El gran viaje con los algoritmos genéticos, lo empecé desarrollando un algoritmo basado en una interfaz la cual sólo usaba dos cosas, población e individuo. A medida que me adentré en el mundo de los algoritmos evolutivos, tuve que ir desechando estructuras para perfeccionarlas, algo que he hecho durante todo el curso, para acabar teniendo un framework creado por mí mismo. Entre los cambios realizados, se encuentran:

- Modificación del esquema de algoritmo genético que ha acabado dividido en operadores, además se ha cambiado la forma en que se llena la población, y se ha permitido disponer de los diferentes tipos de algoritmos genéticos: generacionales y estacionarios y sus subtipos. También se ha creado cada tipo de modelo de islas, aunque fueron abandonadas para dedicar tiempo a otras cosas y porque los genéticos ya estaban muy completos. Pero aún se conservan. Gracias a esto, pueden cambiar su implementación incluso en tiempo de ejecución. Para finalizar, se ha incorporado el *memetismo* como una característica dentro del esquema, el cual, varía su comportamiento incorporando una búsqueda local.

- La población como estructura ha cambiado también, y he creado los tipos mencionados en esa parte: generacionales, ordenadas, desordenadas, ordenables, y los tipos de estacionarias.

- Variar la estructura del individuo (solución), creando una mayor flexibilidad e ir creando más clases de individuos. Para estos individuos he creado verificadores y correctores, que incluso han podido ser usados para verificar la solución de otros algoritmos. Y en el caso del problema de la Mochila 0-1, he creado penalizadores de varios tipos.

- Modificación de la construcción de la población inicial para crear una población inicial rica, que barra la región de soluciones factibles, y por lo tanto creando una diversidad genética inicial.

- Además, entre los diversos cambios, ha habido una investigación de gran cantidad de operadores de todo tipo, implementando bastantes, aunque al final solo han sido nombrados los principales.

He realizado además una serie de adaptaciones, y he llevado a cabo optimizaciones para la mayoría de las metaheurísticas.

Finalmente, la parte del sistema no relacionada con la parte web, la he organizado totalmente, añadiendo al sistema los algoritmos que mi compañero ha implementado, a veces adaptándolo o creando decoradores (clases que envuelven a otras para adaptar), creando las pruebas necesarias, metiendo todo en un servicio de aplicación, creando ejecuciones automáticas, etc…

No podemos olvidarnos de la parte web, para la cual ha sido necesaria una investigación de tecnologías web, como *applets* o *servlets*. Mi compañero conocía *Struts*, por lo que estudié un poco esa tecnología, pero pronto fue desechada como las anteriores.
Finalmente, ambos investigamos el método para crear unos servicios web que nos proporcionan la funcionalidad algorítmica de la que disponemos actualmente. Sabiendo cómo crearlos, tuve que investigar ciertos protocolos, entre ellos, JAX WS, y aprovechando esto y los conocimientos en servicios web del curso anterior, he creado de manera total el servicio web del problema de la mochila y el del TSP. Lo he creado casi todo, a excepción de algunas cosas, como, por ejemplo, usar una estructura de nombres que mi compañero añadió porque necesitaba y la creación de mapas que usaba una factoría creada por mí, método el cual fue sustituido por otro método del servicio de aplicación que usa un método más flexible. Ambos servicios web, se han basado en gran parte, en los servicios de aplicación que he implementado lo que ha facilitado mucho la creación de estos.

Para hacer más flexible la construcción de la interfaz web, la he dividido en partes creando la parte para elegir los algoritmos y las tablas de los parámetros de estos y junto con mi compañero la cabecera y el pie. A su vez, creé en su totalidad, la interfaz web de la mochila con sus elementos, (sin incluir la última colocación con materialize), destacando la representación de los ítems y su correspondiente investigación, y creé todos los scripts y estructuras necesarios para su funcionamiento
La parte del TSP, como podemos ver, era más compleja, por lo que fue realizada a medias, y para ello estudié la API de Google Maps, y entre otros, métodos, la creación de los mapas, el uso de distancias, la matriz de distancias, marcadores, geolocalizadores, rutas, etc. Dentro del TSP, me encargué del estudio inicial de la importación de los mapas de Google y su debida configuración con scripts. También me he encargado de crear un mapa, a partir de los datos de las distancias entre las ciudades provenientes de Google.

Para la parte estética de la interfaz, he estudiado métodos, entre ellos el uso de plantillas, de las cuales, he revisado gran cantidad. Y aunque inicialmente usamos una plantilla, realicé junto con mi compañero una mejora visual de la mochila con Materialize. Mi compañero se encargó de la mejora visual de TSP, y ciertos elementos del TSP, fueron reutilizados en mochila. Apliqué mis conocimientos en HTML5 para crear una web que verifique los datos insertados por el usuario, además, con JavaScript hice comprobaciones y correcciones para evitar entradas incorrectas.
Después de una reunión con nuestro tutor, nos aconsejó mejoras con las cuales él estaría satisfecho. Para realizar todas estas mejoras, tuve que modificar los servicios web tanto de mochila como de TSP. Dichas mejoras eran las siguientes:

- Permitir cierto manejo de ficheros para poder cargar instancias por defecto de los problemas, y, además, poder crear instancias nuevas y almacenarlas sin tener que repetir el laborioso proceso de creación manual por parte del usuario.

- Mostrar en la interfaz un desarrollo de cómo evoluciona la solución en las metaheurísticas, de manera que tuve que partir el algoritmo, indicando el progreso.

- Un historial, en el cual se va almacenando los resultados

- Una mejora visual de la interfaz, como hemos explicado ya arriba.

Tiempo después, tuve que modificar las páginas incluyendo, tanto elementos nuevos, como los scripts, y los servicios web para poder realizar las mejoras mencionadas anteriormente.

La parte de gestión y organización, la empecé de manera muy básica, al hacer una pequeña planificación temporal con unas estimaciones, asignando un coste fijo a cada problema y a cada metaheurística. Estimé un mes para realizar la interfaz y un mes para la memoria, empezando con una investigación inicial y siguiendo con una investigación sobre adaptación, de manera que ciertas tareas estaban pensadas para realizarse de manera simultánea. Aunque esta planificación se ha ido modificando según el progreso que ha habido, ha servido como inicio para hacernos una idea. Para llevar un seguimiento de mi trabajo personal, durante todo el curso he realizado un cuaderno de bitácora (diario) explicando cada día lo que he hecho, los problemas que he tenido, cómo los he solucionado si he podido, y si se me ha ocurrido una solución, aunque no haya podido implementarla el día en cuestión, la he indicado para llevarlo a cabo otro día, y cuántas horas he trabajado. Gracias a esto, puedo calcular que he estado dedicando durante la realización de este proyecto, una media aproximada, de 7 horas, la

mayoría del tiempo he estado una media de 8 horas, pero ha habido periodos en los que he trabajado 4 o 5 horas y algunos, en los que he estado 6 horas. Cabe destacar que el máximo alcanzado en un día ha sido 10 horas.
Para lograr una gestión correcta del contenido, he implantado el uso de herramientas de control de versiones. A pesar de este control, ha habido una serie de problemas que he podido resolver gracias a estas herramientas y hemos podido organizar las versiones y recuperar elementos necesarios que se podrían haber perdido.

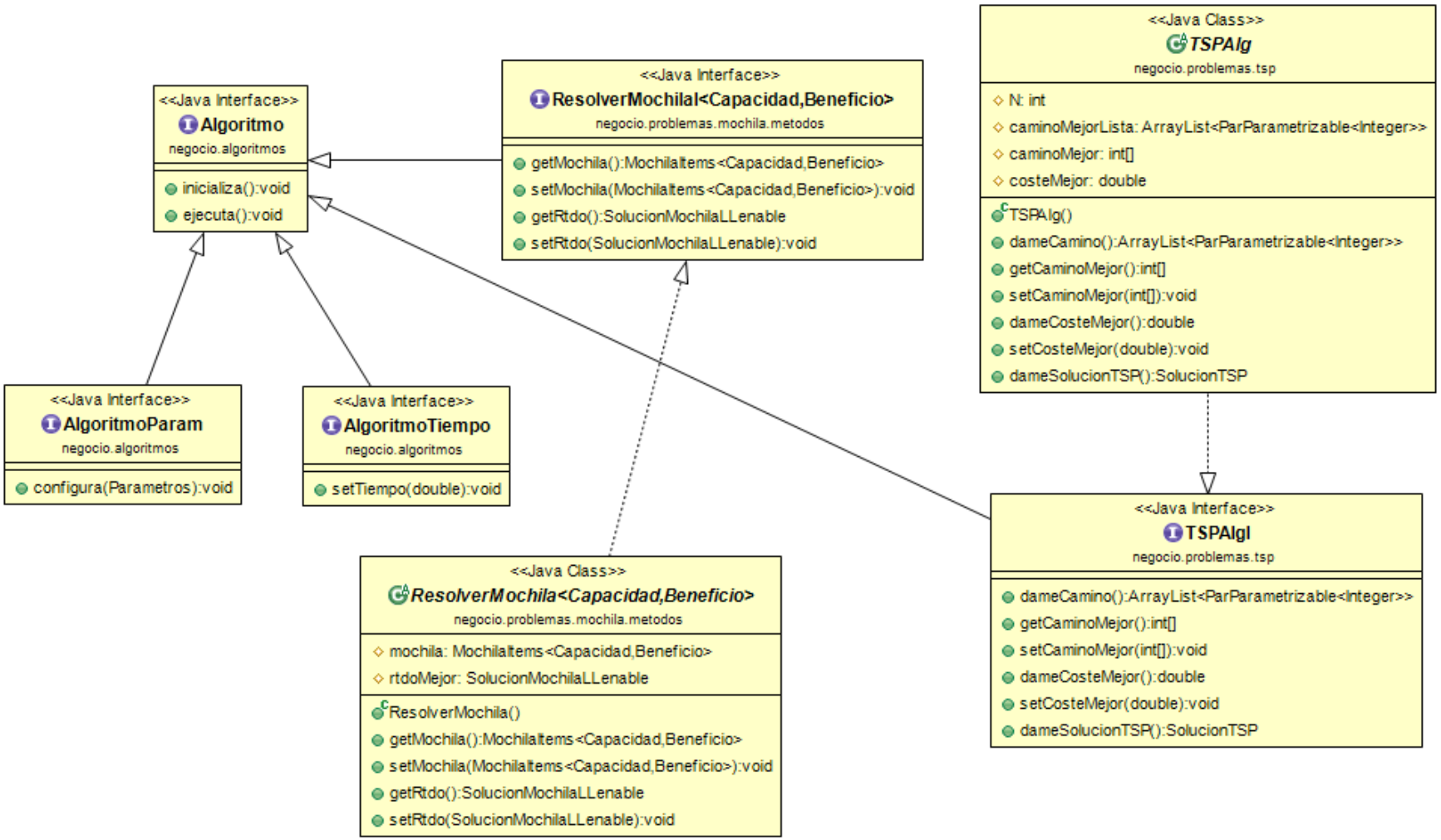
El control de calidad, se ha intentado realizar de manera continua, aunque ciertas tareas, por su naturaleza, se han llevado a cabo de manera interrumpida. He ido haciendo de manera continua pruebas y revisiones de la aplicación para evitar todo posible fallo y que este se propague. Además, he realizado una revisión de los documentos, tanto de mi parte como la de mi compañero. No me he conformado con un funcionamiento decente, me he dedicado a perfeccionar el sistema lo máximo posible y para ello he mejorado los algoritmos genéticos, y realizado optimizaciones tanto en ACO/ACE, como en SA o GRASP.

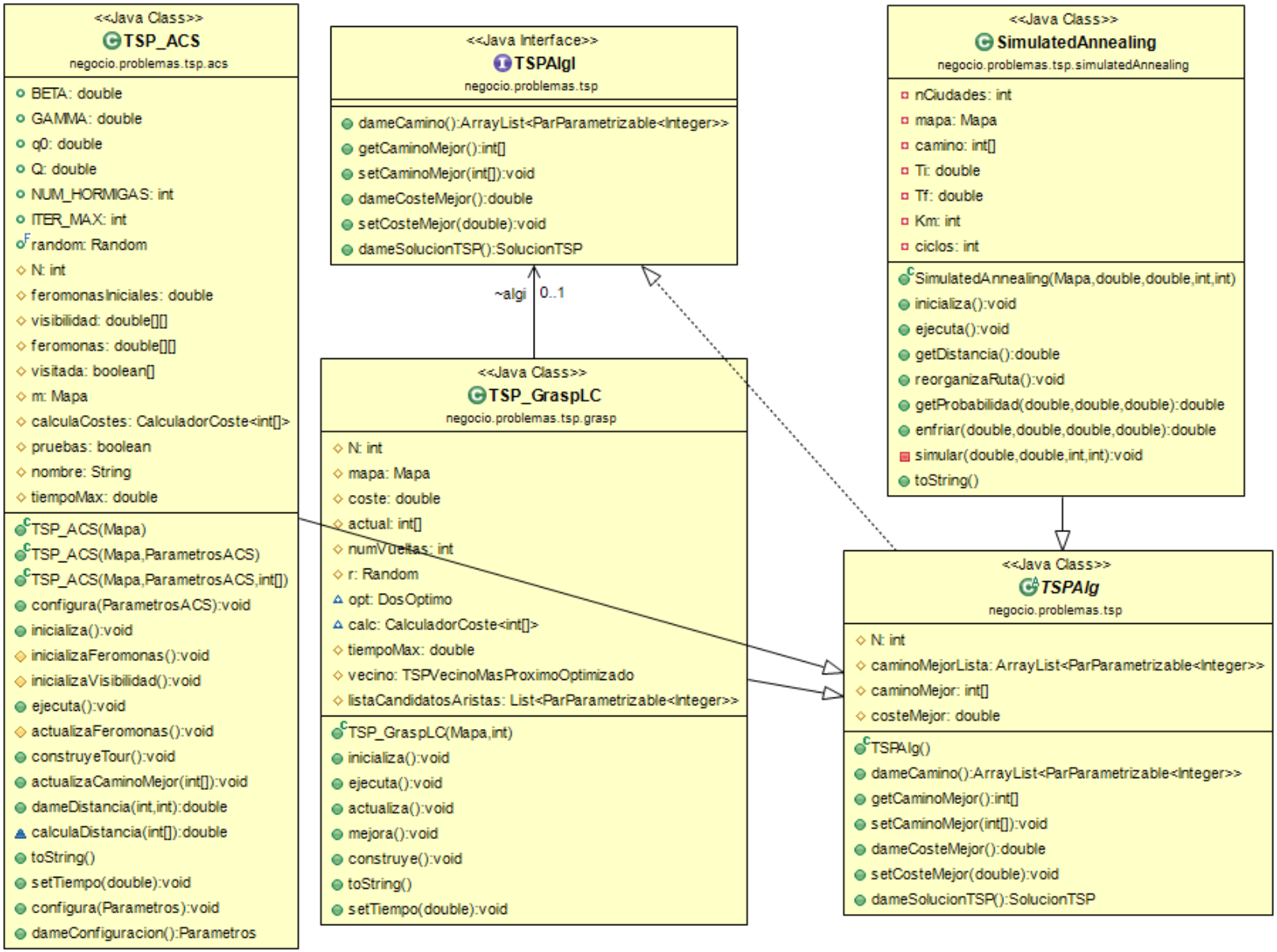
Para comprobar el funcionamiento y la eficacia de los algoritmos, desarrollé pruebas y ejecuciones tanto individuales como globales para compararlos entre todos. Al inicio, para el TSP creé mapas de pruebas con como mucho 20 ciudades y un mapa aleatorio. Además, para ellos creé factorías para encapsular los tipos de mapas y encapsular la creación de mapas en una clase, de manera que se separa todo por capas o paquetes. Más tarde creé mapas más específicos con diversas implementaciones, entre ellos hay algunos que se leen por fichero. Una vez con los elementos necesarios, mapa y algún algoritmo, pude crear un automatismo para probar cada uno de ellos por separado y todos juntos. De manera inicial, las pruebas las realizaba solo fijándome en su coste. Más tarde creé unas clases para organizar los ficheros y funcionalidades (p. ej. encontrar los ficheros que tengan menos de x ciudades, etc.) y creé formas para calcular el tiempo usado por cada algoritmo para ejecutarse. Debido a que cada vez había más datos y más pruebas y ejecuciones, creé entre ellas unas clases para encapsular los resultados del TSP incluyendo coste y tiempo, de manera que una vez ejecutados, se pueden ordenar y además comparar la eficacia de todos con el mejor, para así poder establecer el porcentaje de eficacia relativa del algoritmo. Para sacar los resultados a un fichero investigué algunas formas y al final pensé que lo más sencillo era el formato .csv, el cual podía ser abierto por Excel y con el que hemos podido crear gran cantidad de gráficos. Además, creé unas clases para poder estructurar los resultados, crear por instancia, para comparar todos los algoritmos entre ellos en una misma instancia y para comparar un mismo algoritmo en las distintas instancias del problema. Además, también he realizado pruebas según la configuración de los parámetros de los algoritmos, aunque a veces ha podido ser una ardua tarea por la gran cantidad de configuraciones posibles. Algo más complejo, ha sido realizar pruebas automáticas según los operadores de los algoritmos genéticos, de manera que he tenido que crear *"builders"* para crear algoritmos genéticos con las diferentes combinaciones de operadores disponibles, y pruebas especializadas para ello. Aunque parece algo simple, ha habido varios problemas a la hora de realizar todas estas pruebas, los cuales me han ocupado bastante tiempo, pero a base de investigar los pude solventar. De entre las cosas que estudié para crear las pruebas y ejecuciones, fue Java 8 con lo que, por ejemplo, he podido agrupar ciertos resultados para organizarlos y realizar cálculos. Gracias también a Java 8 he podido reinterpretar resultados almacenados en ficheros y realizar cálculos tales como medias de eficacia. Como colofón, he creado una clase para ejecutar un fichero de manera automática, de tal manera que cada línea se componga de partes separadas por comas las cuales son el problema, el algoritmo y sus parámetros, y de esta manera, se pueden realizar las ejecuciones de cada línea almacenando los resultados en un fichero externo, de manera que el usuario no necesite ejecutarlos uno a uno.

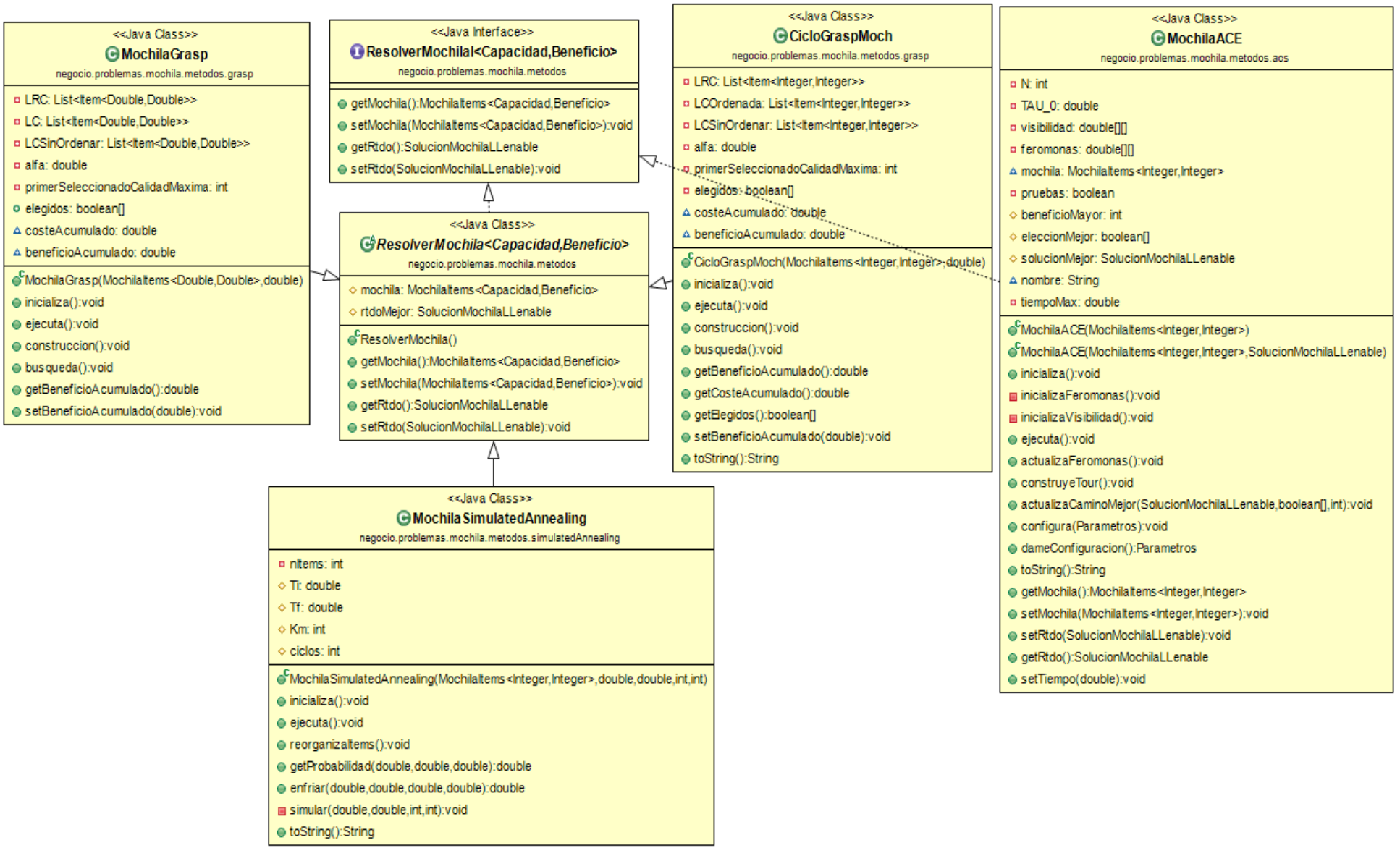
Para finalizar, he participado de manera intensa en la memoria, participando de manera parcial o total en cada apartado de esta, elaborando tareas de organización, redacción, corrección y análisis.

Por último, yo también quiero agradecer a mi compañero por su trabajo. Sus aportaciones ya sean pocas o muchas han sido siempre importantes para poder sacar adelante este trabajo.

Apéndice A: diagrama de clases para los algoritmos



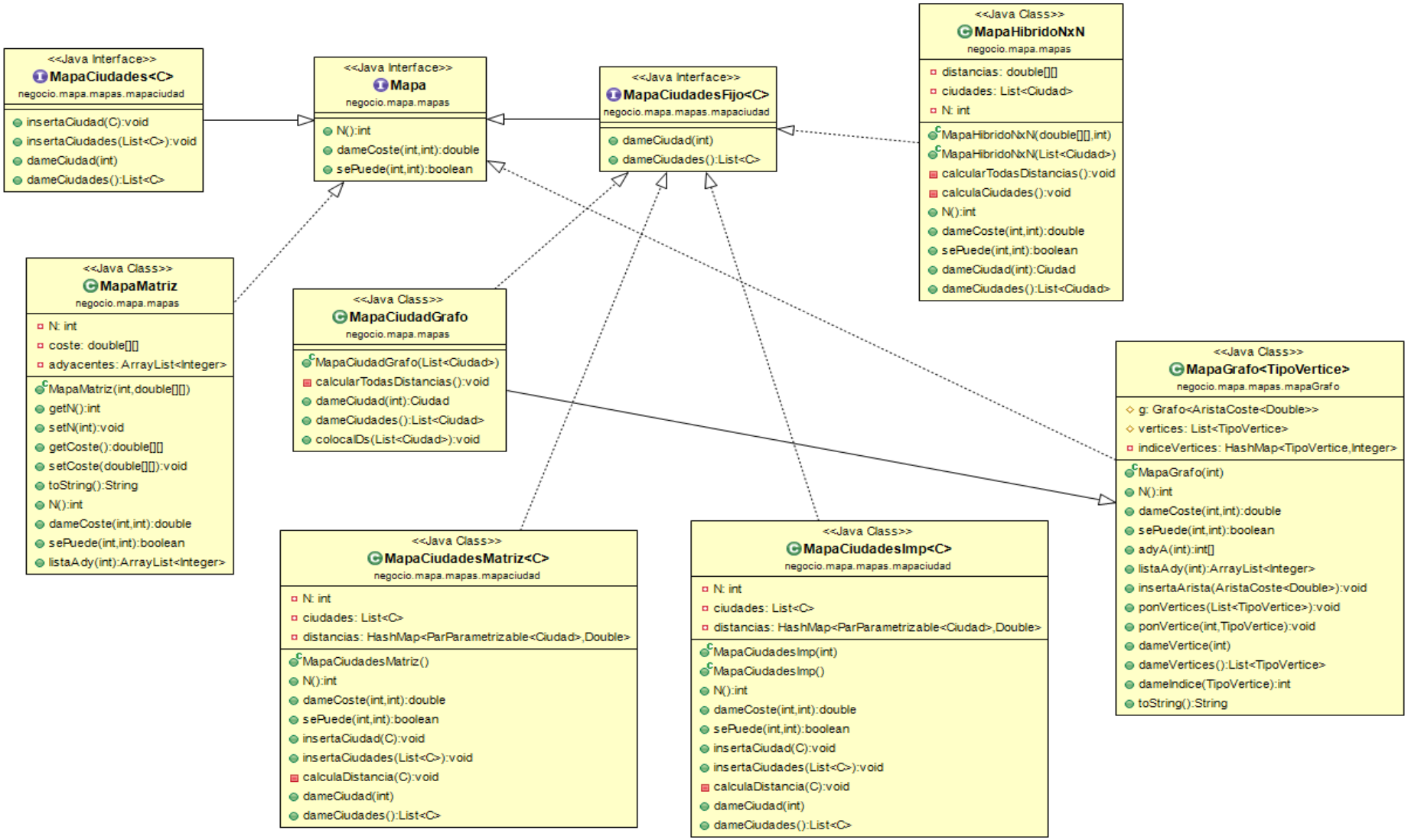

<<Java Class>>
TSP_ACS
negocio.problemas.tsp.acs
BETA: double
GAMMA: double
q0: double
Q: double
NUM_HORMIGAS: int
ITER_MAX: int
random: Random
N: int
feromonasIniciales: double
visibilidad: double[][]
feromonas: double[][]
visitada: boolean[]
m: Mapa
calculaCostes: CalculadorCoste<int[]>
pruebas: boolean
nombre: String
tiempoMax: double
TSP_ACS(Mapa)
TSP_ACS(Mapa,ParametrosACS)
TSP_ACS(Mapa,ParametrosACS,int[])
configura(ParametrosACS):void
inicializa():void
inicializaFeromonas():void
inicializaVisibilidad():void
ejecuta():void
actualizaFeromonas():void
construyeTour():void
actualizaCaminoMejor(int[]):void
dameDistancia(int,int):double
calculaDistancia(int[]):double
toString()
setTiempo(double):void
configura(Parametros):void
dameConfiguracion():Parametros
<<Java Interface>>
TSPAlgI
negocio.problemas.tsp
dameCamino():ArrayList<ParParametrizable<Integer>>
getCaminoMejor():int[]
setCaminoMejor(int[]):void
dameCosteMejor():double
setCosteMejor(double):void
dameSolucionTSP():SolucionTSP
~algi
0..1
<<Java Class>>
TSP_GraspLC
negocio.problemas.tsp.grasp
N: int
mapa: Mapa
coste: double
actual: int[]
numVueltas: int
r: Random
opt: DosOptimo
calc: CalculadorCoste<int[]>
tiempoMax: double
vecino: TSPVecinoMasProximoOptimizado
listaCandidatosAristas: List<ParParametrizable<Integer>>
TSP_GraspLC(Mapa,int)
inicializa():void
ejecuta():void
actualiza():void
mejora():void
construye():void
toString()
setTiempo(double):void
<<Java Class>>
SimulatedAnnealing
negocio.problemas.tsp.simulatedAnnealing
nCiudades: int
mapa: Mapa
camino: int[]
Ti: double
Tf: double
Km: int
ciclos: int
SimulatedAnnealing(Mapa,double,double,int,int)
inicializa():void
ejecuta():void
getDistancia():double
reorganizaRuta():void
getProbabilidad(double,double,double):double
enfriar(double,double,double,double):double
simular(double,double,int,int):void
toString()
<<Java Class>>
TSPAlg
negocio.problemas.tsp
N: int
caminoMejorLista: ArrayList<ParParametrizable<Integer>>
caminoMejor: int[]
costeMejor: double
TSPAlg()
dameCamino():ArrayList<ParParametrizable<Integer>>
getCaminoMejor():int[]
setCaminoMejor(int[]):void
dameCosteMejor():double
setCosteMejor(double):void
dameSolucionTSP():SolucionTSP


<<Java Class>>
MochilaGrasp
negocio.problemas.mochila.metodos.grasp
LRC: List<Item<Double,Double>>
LC: List<Item<Double,Double>>
LCSinOrdenar: List<Item<Double,Double>>
alfa: double
primerSeleccionadoCalidadMaxima: int
elegidos: boolean[]
costeAcumulado: double
beneficioAcumulado: double
MochilaGrasp(MochilaItems<Double,Double>,double)
inicializa():void
ejecuta():void
construccion():void
busqueda():void
getBeneficioAcumulado():double
setBeneficioAcumulado(double):void
<<Java Interface>>
ResolverMochila<Capacidad,Beneficio>
negocio.problemas.mochila.metodos
getMochila():MochilaItems<Capacidad,Beneficio>
setMochila(MochilaItems<Capacidad,Beneficio>):void
getRtdo():SolucionMochilaLLenable
setRtdo(SolucionMochilaLLenable):void
<<Java Class>>
ResolverMochila<Capacidad,Beneficio>
negocio.problemas.mochila.metodos
mochila: MochilaItems<Capacidad,Beneficio>
rtdoMejor: SolucionMochilaLLenable
ResolverMochila()
getMochila():MochilaItems<Capacidad,Beneficio>
setMochila(MochilaItems<Capacidad,Beneficio>):void
getRtdo():SolucionMochilaLLenable
setRtdo(SolucionMochilaLLenable):void
<<Java Class>>
MochilaSimulatedAnnealing
negocio.problemas.mochila.metodos.simulatedAnnealing
nItems: int
Ti: double
Tf: double
Km: int
ciclos: int
MochilaSimulatedAnnealing(MochilaItems<Integer,Integer>,double,double,int,int)
inicializa():void
ejecuta():void
reorganizaItems():void
getProbabilidad(double,double,double):double
enfriar(double,double,double,double):double
simular(double,double,int,int):void
toString():String
<<Java Class>>
CicloGraspMoch
negocio.problemas.mochila.metodos.grasp
LRC: List<Item<Integer,Integer>>
LCOrdenada: List<Item<Integer,Integer>>
LCSinOrdenar: List<Item<Integer,Integer>>
alfa: double
primerSeleccionadoCalidadMaxima: int
elegidos: boolean[]
costeAcumulado: double
beneficioAcumulado: double
CicloGraspMoch(MochilaItems<Integer,Integer>,double)
inicializa():void
ejecuta():void
construccion():void
busqueda():void
getBeneficioAcumulado():double
getCosteAcumulado():double
getElegidos():boolean[]
setBeneficioAcumulado(double):void
toString():String
<<Java Class>>
MochilaACE
negocio.problemas.mochila.metodos.acs
N: int
TAU_0: double
visibilidad: double[][]
feromonas: double[][]
mochila: MochilaItems<Integer,Integer>
pruebas: boolean
beneficioMayor: int
eleccionMejor: boolean[]
solucionMejor: SolucionMochilaLLenable
nombre: String
tiempoMax: double
MochilaACE(MochilaItems<Integer,Integer>)
MochilaACE(MochilaItems<Integer,Integer>,SolucionMochilaLLenable)
inicializa():void
inicializaFeromonas():void
inicializaVisibilidad():void
ejecuta():void
actualizaFeromonas():void
construyeTour():void
actualizaCaminoMejor(SolucionMochilaLLenable,boolean[],int):void
configura(Parametros):void
dameConfiguracion():Parametros
toString():String
getMochila():MochilaItems<Integer,Integer>
setMochila(MochilaItems<Integer,Integer>):void
setRtdo(SolucionMochilaLLenable):void
getRtdo():SolucionMochilaLLenable
setTiempo(double):void

La interfaz principal es la interfaz *Algoritmo*. Esta contiene dos métodos que deben ser implementados por aquellas clases que implementen dicha interfaz. Por un lado, tenemos al método *inicializa*, que está pensado para inicializar los elementos y los parámetros del algoritmo que vamos a ejecutar, y por otro lado tenemos al método *ejecuta*, que se encarga de ejecutar el algoritmo en busca de alguna solución.

Tenemos dos interfaces más que heredan de la anterior: la interfaz *ResolverMochilaI*, específica para los algoritmos que resuelven el problema de la Mochila 0-1, y la interfaz *TSPAlg*, específica para los algoritmos del TSP.

Apéndice B: diagrama de clases para los mapas



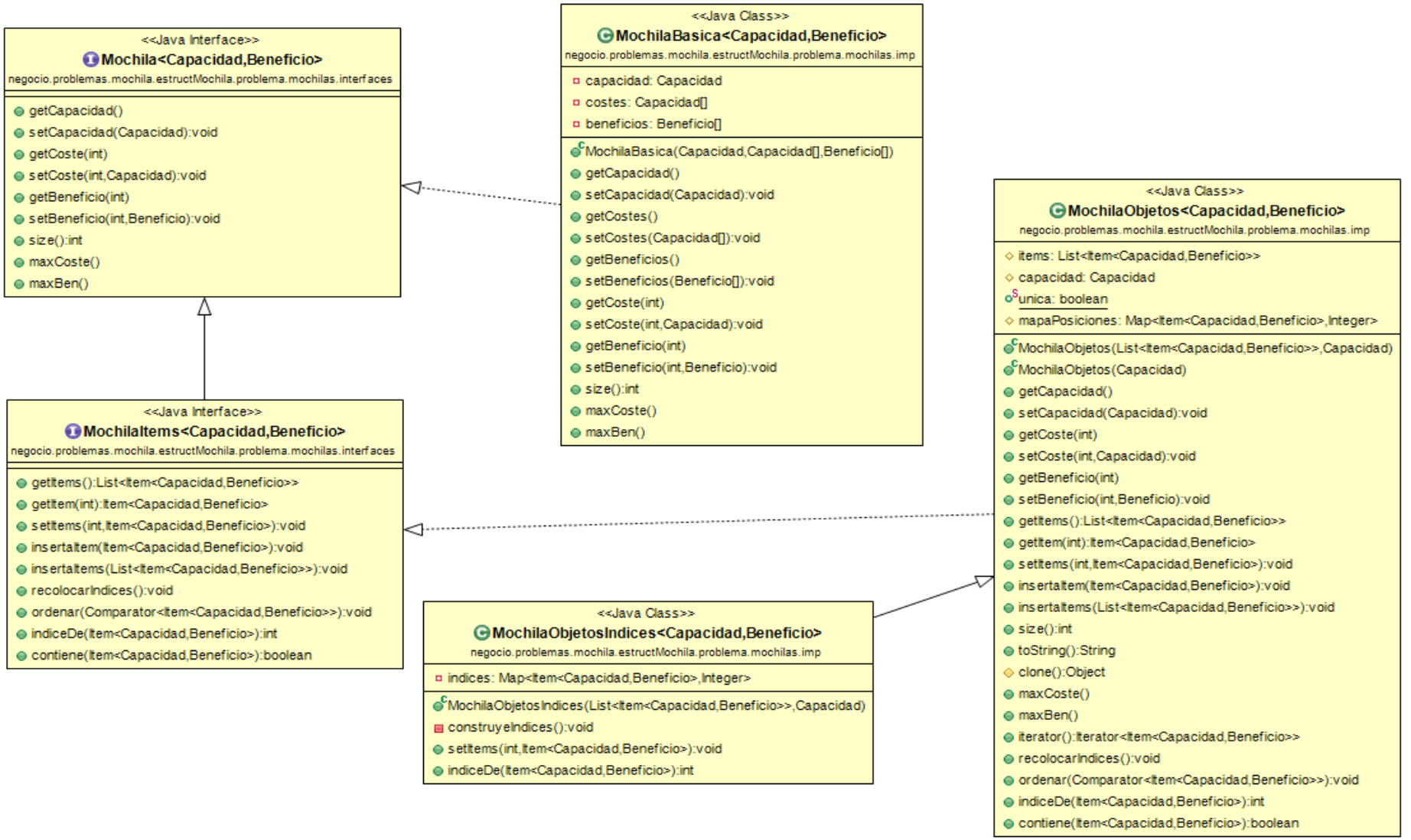
Sin entrar en tanto detalle, este es el diagrama de clases que contiene a los tipos de mapas ya presentados en la sección 4.1.1. y que sirven para la resolución del problema del TSP. La interfaz principal es la interfaz "Mapa", cuyos métodos principales y que deben ser implementados por cada uno de los tipos de mapa son:

- *N,* que devuelve el número de ciudades del mapa en cuestión.

- *dameCoste*, que, dadas dos ciudades cualesquiera, devuelve la distancia que las separa.

- *sePuede*, que, dadas dos ciudades cualesquiera, comprueba si estas están conectadas.

Apéndice C: diagrama de clases para las mochilas



Al igual que ocurre con los mapas para el problema del TSP, los tipos de mochila para la resolución del problema de la Mochila 0-1 ya fueron presentados en el capítulo 4. En este diagrama de clases la interfaz principal es la interfaz "Mochila", la cual contiene un total de 9 métodos a ser implementados por cada uno de los tipos de mochila:

- *getCapacidad*, que devuelve la capacidad de la mochila del problema.

- *setCapacidad*, que modifica la capacidad de la mochila del problema.

- *getCoste*, que devuelve el coste de un determinado ítem.

- *setCoste*, que modifica el coste de un determinado ítem

- *getBeneficio*, que devuelve el beneficio de un determinado ítem.

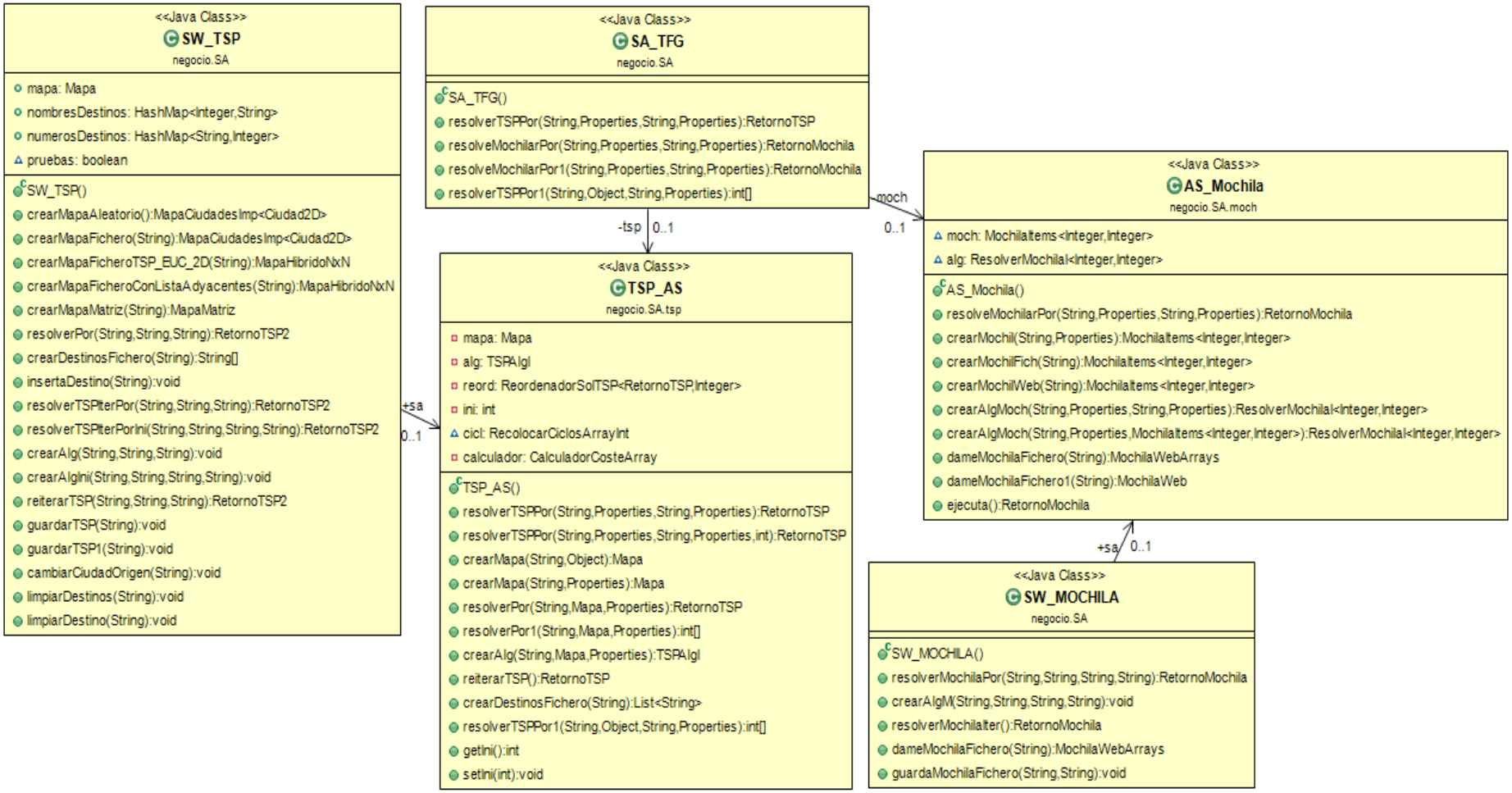
- *setBeneficio*, que modifica el beneficio de un determinado ítem.

- *Size*, que devuelve el número de ítems.

- *maxCoste*, que devuelve el máximo coste (peso) que se podría acumular si la solución del problema incluyese a todos los ítems.

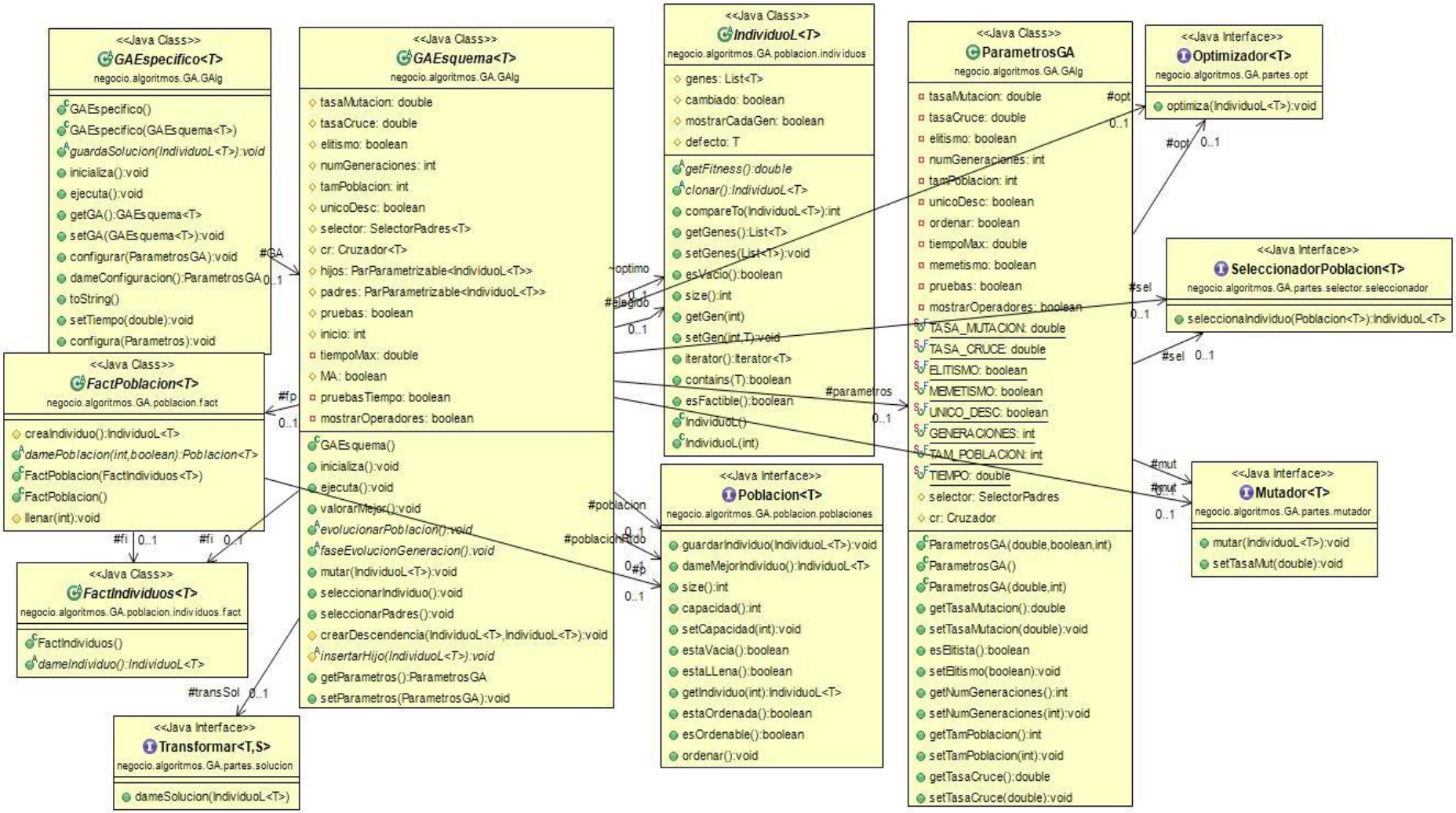
- *MaxBen,* que devuelve el máximo beneficio que se podría acumular si la solución del problema incluyese a todos los ítems.

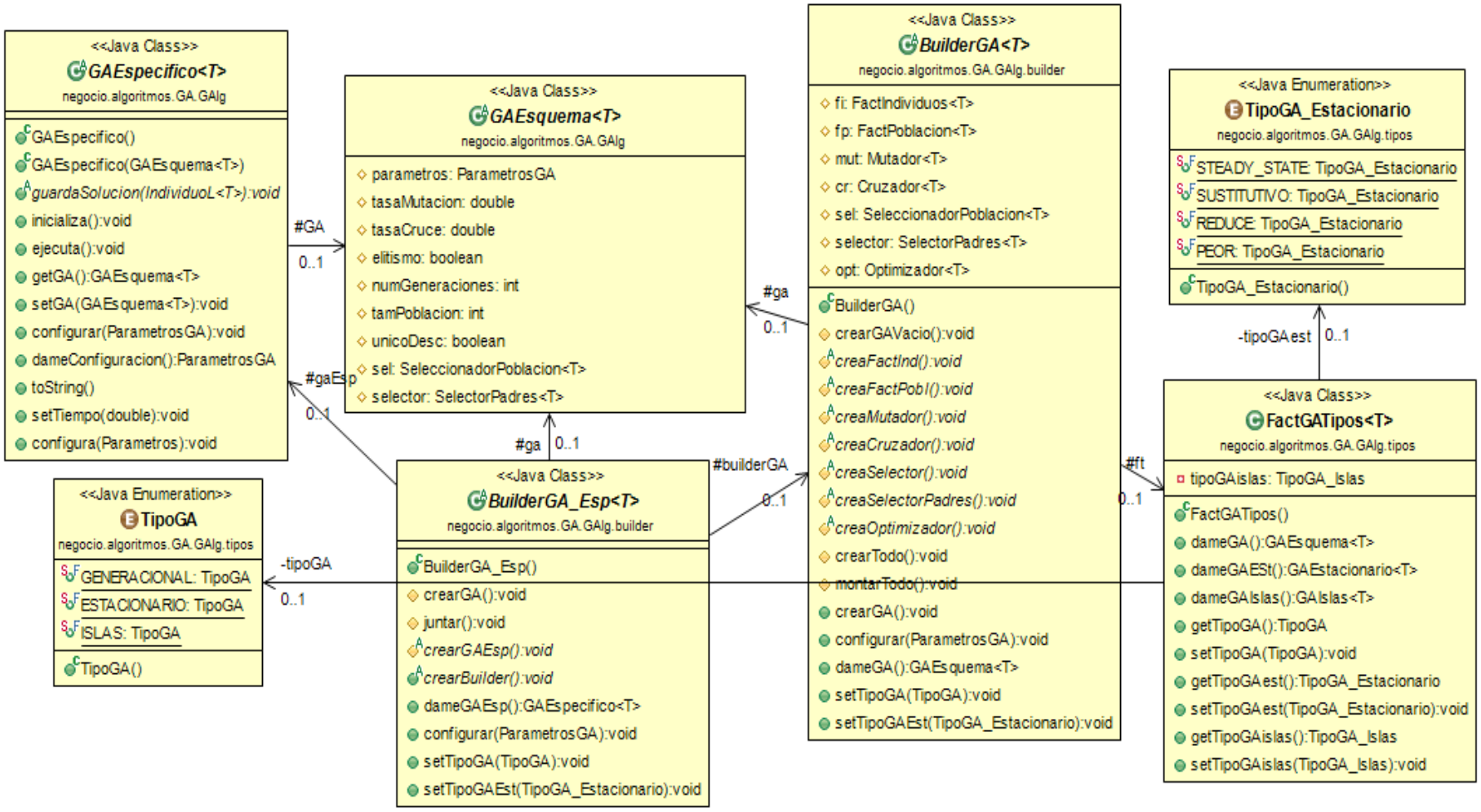
Apéndice D: diagrama de clases para los servicios web

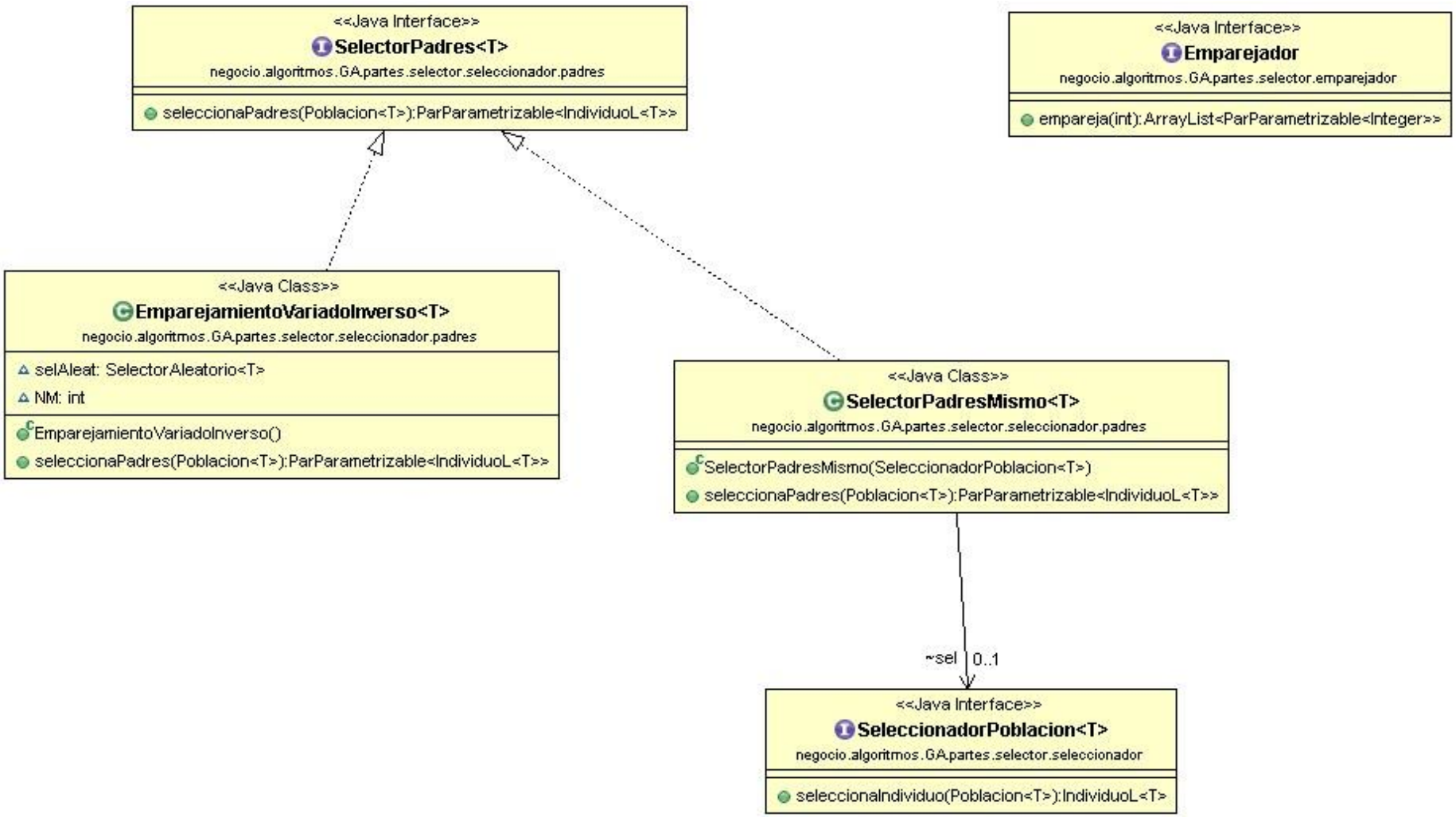


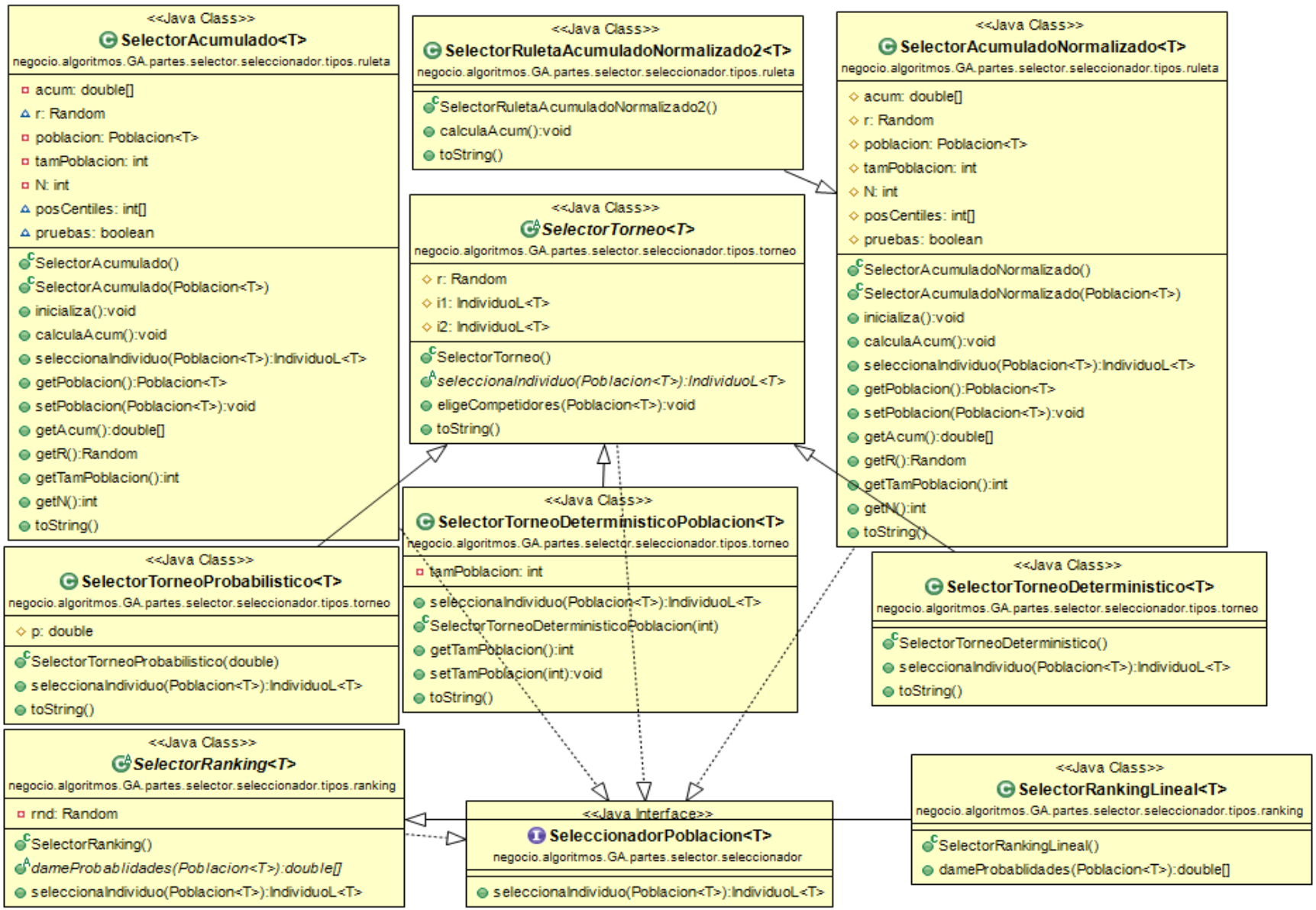
Basándonos en el patrón de diseño *Application Service*, hemos implementado una clase *SA_TFG*, que engloba todas las acciones, tanto del TSP, como del problema de la Mochila 0-1 y que componen la lógica de negocio de nuestro sistema. Partiendo de esta base, pudimos implementar las clases *SW_TSP* y *SW_MOCHILA*, las cuales contienen los servicios web para la resolución de ambos problemas.

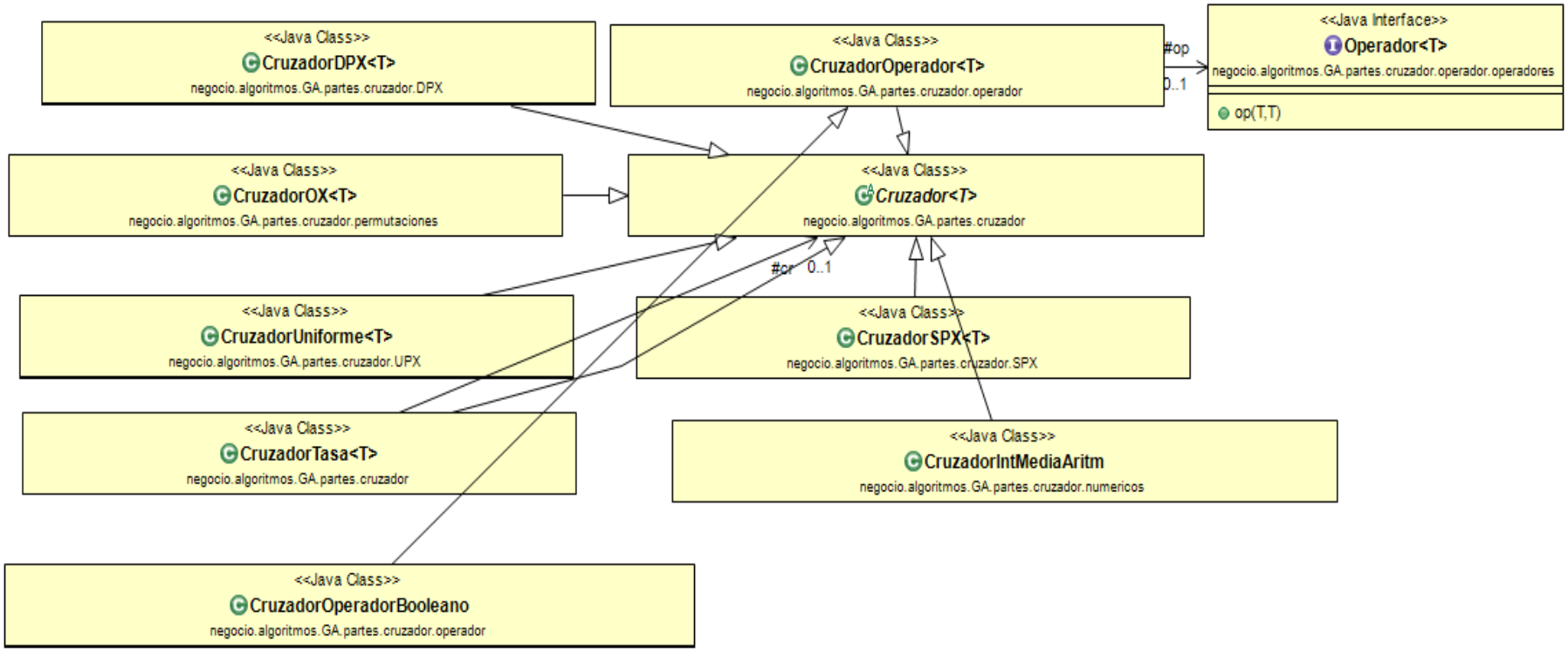
Apéndice E: diagrama de clases para el framework de los algoritmos genéticos

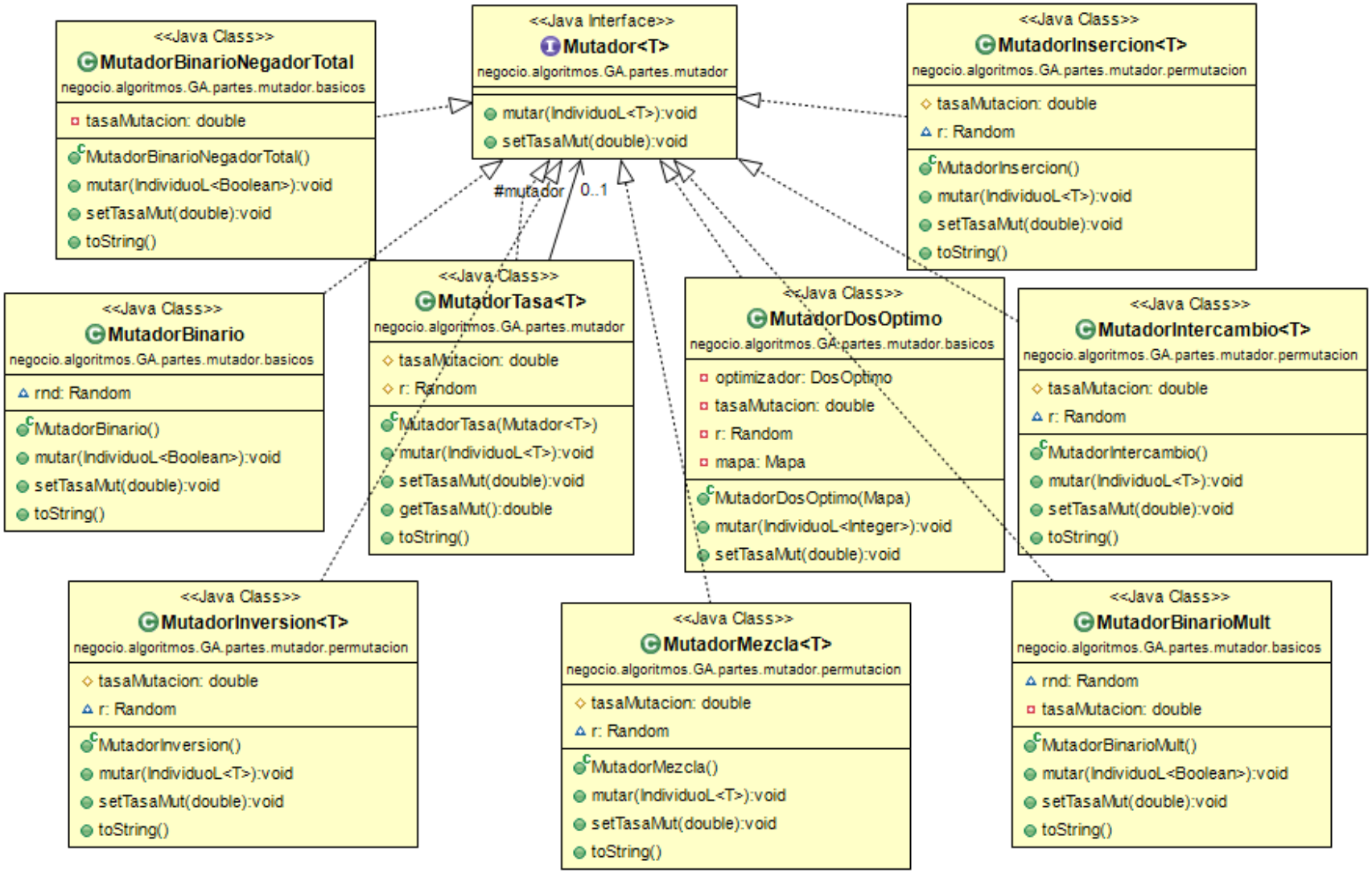


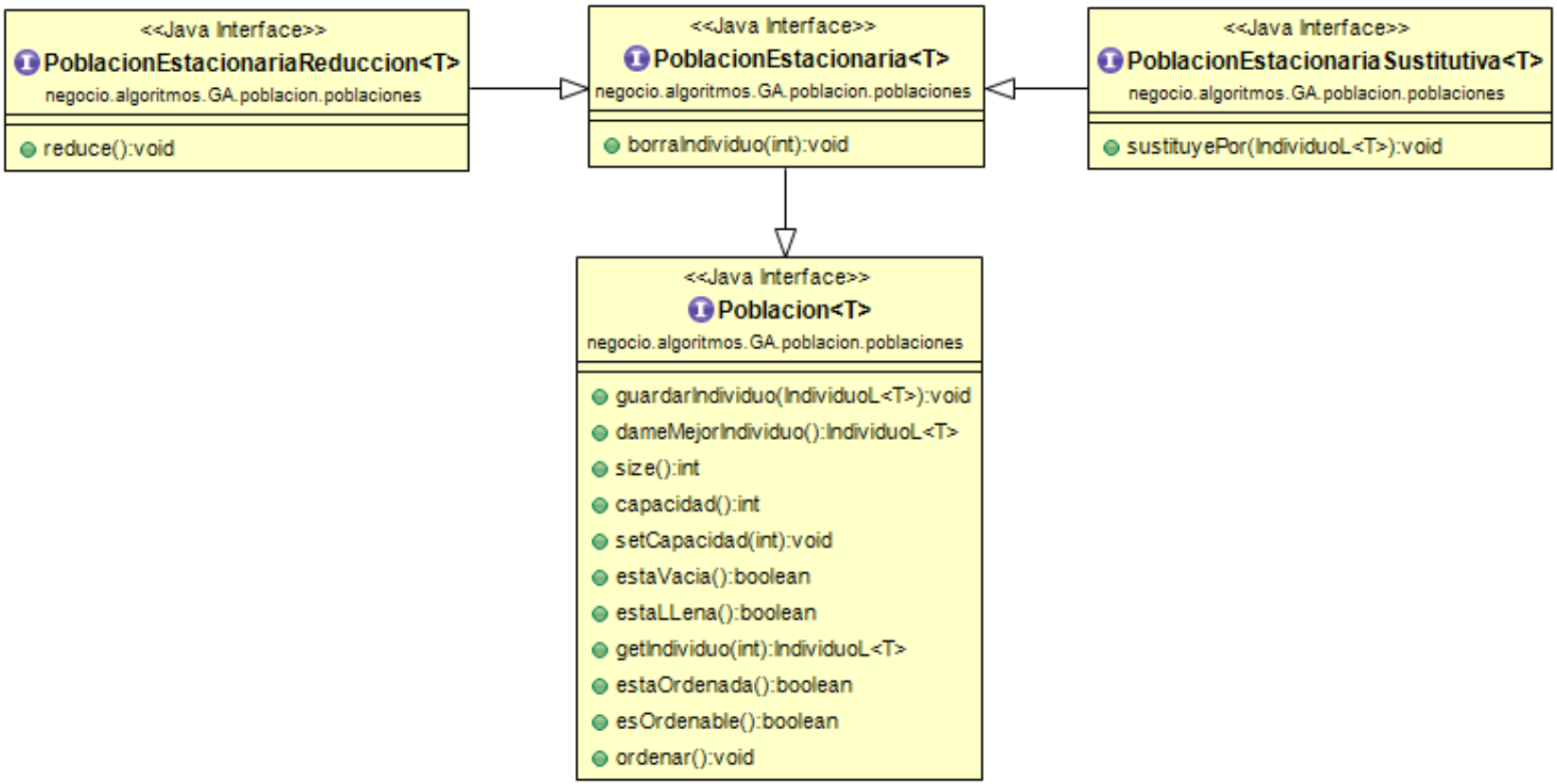

<<Java Class>>
GAEspecifico<T>
negocio.algoritmos.GA.GAlg
GAEspecifico()
GAEspecifico(GAEsquema<T>)
guardaSolucion(IndividuoL<T>):void
inicializa():void
ejecuta():void
getGA():GAEsquema<T>
setGA(GAEsquema<T>):void
configurar(ParametrosGA):void
dameConfiguracion():ParametrosGA
toString()
setTiempo(double):void
configura(Parametros):void
#GA
0..1
#gaEsp
0..1
<<Java Class>>
GAEsquema<T>
negocio.algoritmos.GA.GAlg
parametros: ParametrosGA
tasaMutacion: double
tasaCruce: double
elitismo: boolean
numGeneraciones: int
tamPoblacion: int
unicoDesc: boolean
sel: SeleccionadorPoblacion<T>
selector: SelectorPadres<T>
#ga
0..1
<<Java Class>>
BuilderGA<T>
negocio.algoritmos.GA.GAlg.builder
fi: FactIndividuos<T>
fp: FactPoblacion<T>
mut: Mutador<T>
cr: Cruzador<T>
sel: SeleccionadorPoblacion<T>
selector: SelectorPadres<T>
opt: Optimizador<T>
BuilderGA()
crearGAVacio():void
creaFactInd():void
creaFactPobl():void
creaMutador():void
creaCruzador():void
creaSelector():void
creaSelectorPadres():void
creaOptimizador():void
crearTodo():void
montarTodo():void
crearGA():void
configurar(ParametrosGA):void
dameGA():GAEsquema<T>
setTipoGA(TipoGA):void
setTipoGAEst(TipoGA_Estacionario):void
<<Java Enumeration>>
TipoGA_Estacionario
negocio.algoritmos.GA.GAlg.tipos
STEADY_STATE: TipoGA_Estacionario
SUSTITUTIVO: TipoGA_Estacionario
REDUCE: TipoGA_Estacionario
PEOR: TipoGA_Estacionario
TipoGA_Estacionario()
-tipoGAest
0..1
<<Java Class>>
FactGATipos<T>
negocio.algoritmos.GA.GAlg.tipos
tipoGAislas: TipoGA_Islas
FactGATipos()
dameGA():GAEsquema<T>
dameGAESt():GAEstacionario<T>
dameGAIslas():GAIslas<T>
getTipoGA():TipoGA
setTipoGA(TipoGA):void
getTipoGAest():TipoGA_Estacionario
setTipoGAest(TipoGA_Estacionario):void
getTipoGAislas():TipoGA_Islas
setTipoGAislas(TipoGA_Islas):void
#ft
0..1
<<Java Enumeration>>
TipoGA
negocio.algoritmos.GA.GAlg.tipos
GENERACIONAL: TipoGA
ESTACIONARIO: TipoGA
ISLAS: TipoGA
TipoGA()
-tipoGA
0..1
<<Java Class>>
BuilderGA_Esp<T>
negocio.algoritmos.GA.GAlg.builder
BuilderGA_Esp()
crearGA():void
juntar():void
crearGAEsp():void
crearBuilder():void
dameGAEsp():GAEspecifico<T>
configurar(ParametrosGA):void
setTipoGA(TipoGA):void
setTipoGAEst(TipoGA_Estacionario):void
#ga
0..1
#builderGA
0..1

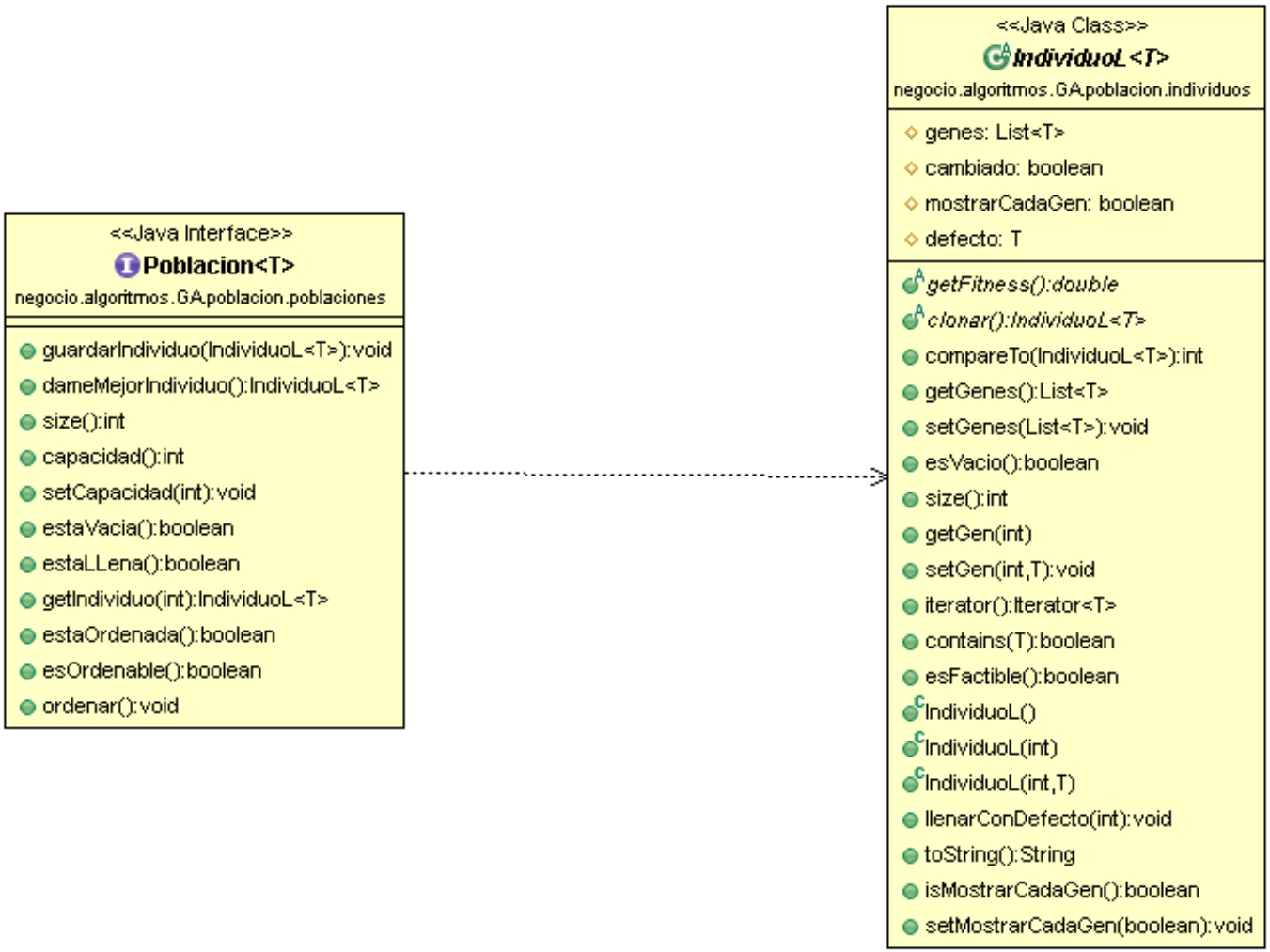

<<Java Interface>>
SelectorPadres<T>
negocio.algoritmos.GA.partes.selector.seleccionador.padres
seleccionaPadres(Poblacion<T>):ParParametrizable<IndividuoL<T>>
<<Java Interface>>
Emparejador
negocio.algoritmos.GA.partes.selector.emparejador
empareja(int):ArrayList<ParParametrizable<Integer>>
<<Java Class>>
EmparejamientoVariadoInverso<T>
negocio.algoritmos.GA.partes.selector.seleccionador.padres
selAleat: SelectorAleatorio<T>
NM: int
EmparejamientoVariadoInverso()
seleccionaPadres(Poblacion<T>):ParParametrizable<IndividuoL<T>>
<<Java Class>>
SelectorPadresMismo<T>
negocio.algoritmos.GA.partes.selector.seleccionador.padres
SelectorPadresMismo(SeleccionadorPoblacion<T>)
seleccionaPadres(Poblacion<T>):ParParametrizable<IndividuoL<T>>
~sel
0..1
<<Java Interface>>
SeleccionadorPoblacion<T>
negocio.algoritmos.GA.partes.selector.seleccionador
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>

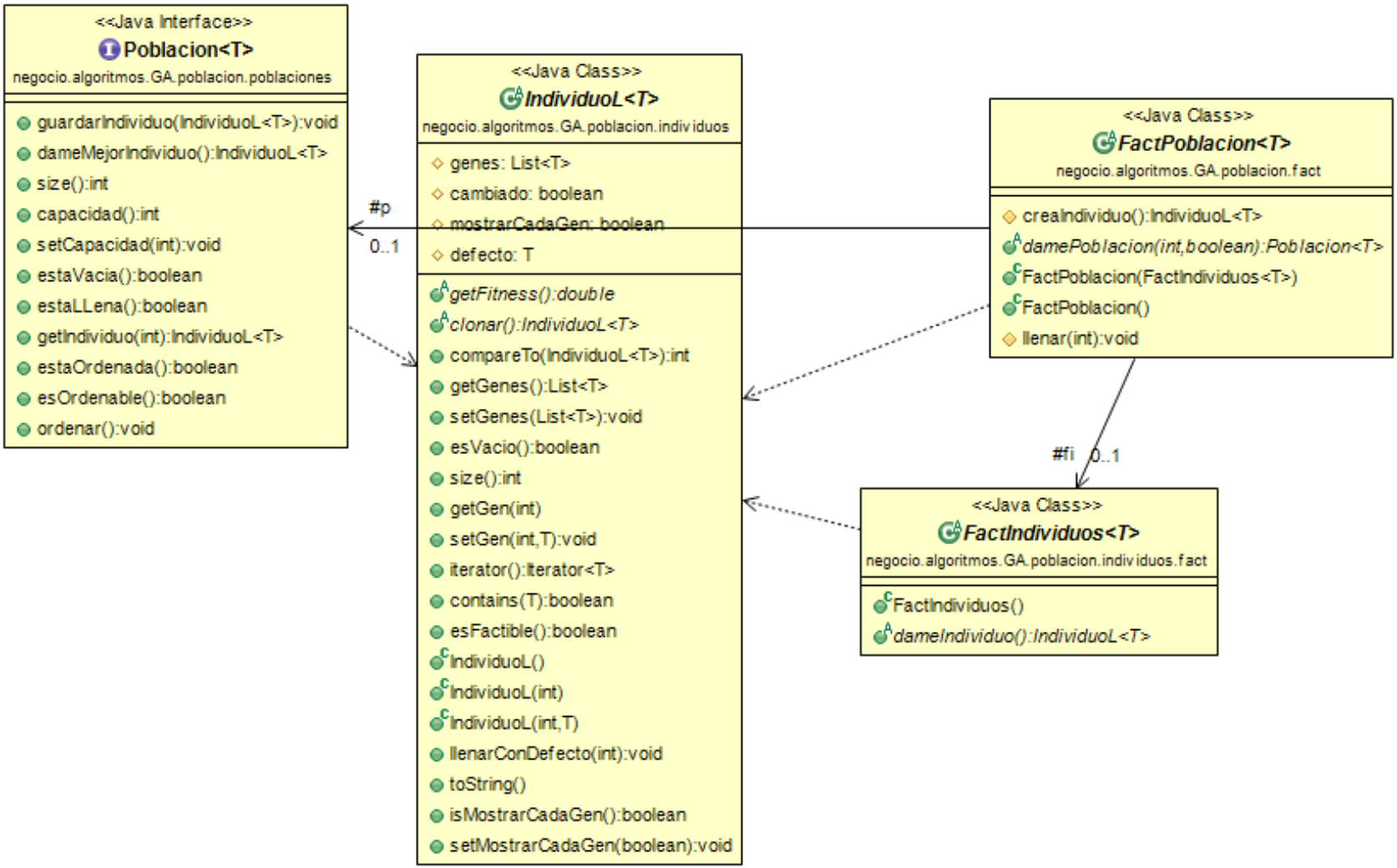

<<Java Class>>
SelectorAcumulado<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.ruleta
acum: double[]
r: Random
poblacion: Poblacion<T>
tamPoblacion: int
N: int
posCentiles: int[]
pruebas: boolean
SelectorAcumulado()
SelectorAcumulado(Poblacion<T>)
inicializa():void
calculaAcum():void
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
getPoblacion():Poblacion<T>
setPoblacion(Poblacion<T>):void
getAcum():double[]
getR():Random
getTamPoblacion():int
getN():int
toString()
<<Java Class>>
SelectorRuletaAcumuladoNormalizado2<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.ruleta
SelectorRuletaAcumuladoNormalizado2()
calculaAcum():void
toString()
<<Java Class>>
SelectorAcumuladoNormalizado<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.ruleta
acum: double[]
r: Random
poblacion: Poblacion<T>
tamPoblacion: int
N: int
posCentiles: int[]
pruebas: boolean
SelectorAcumuladoNormalizado()
SelectorAcumuladoNormalizado(Poblacion<T>)
inicializa():void
calculaAcum():void
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
getPoblacion():Poblacion<T>
setPoblacion(Poblacion<T>):void
getAcum():double[]
getR():Random
getTamPoblacion():int
getN():int
toString()
<<Java Class>>
SelectorTorneo<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.torneo
r: Random
i1: IndividuoL<T>
i2: IndividuoL<T>
SelectorTorneo()
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
eligeCompetidores(Poblacion<T>):void
toString()
<<Java Class>>
SelectorTorneoProbabilistico<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.torneo
p: double
SelectorTorneoProbabilistico(double)
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
toString()
<<Java Class>>
SelectorTorneoDeterministicoPoblacion<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.torneo
tamPoblacion: int
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
SelectorTorneoDeterministicoPoblacion(int)
getTamPoblacion():int
setTamPoblacion(int):void
toString()
<<Java Class>>
SelectorTorneoDeterministico<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.torneo
SelectorTorneoDeterministico()
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
toString()
<<Java Class>>
SelectorRanking<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.ranking
rnd: Random
SelectorRanking()
dameProbablidades(Poblacion<T>):double[]
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
<<Java Interface>>
SeleccionadorPoblacion<T>
negocio.algoritmos.GA.partes.selector.seleccionador
seleccionaIndividuo(Poblacion<T>):IndividuoL<T>
<<Java Class>>
SelectorRankingLineal<T>
negocio.algoritmos.GA.partes.selector.seleccionador.tipos.ranking
SelectorRankingLineal()
dameProbablidades(Poblacion<T>):double[]

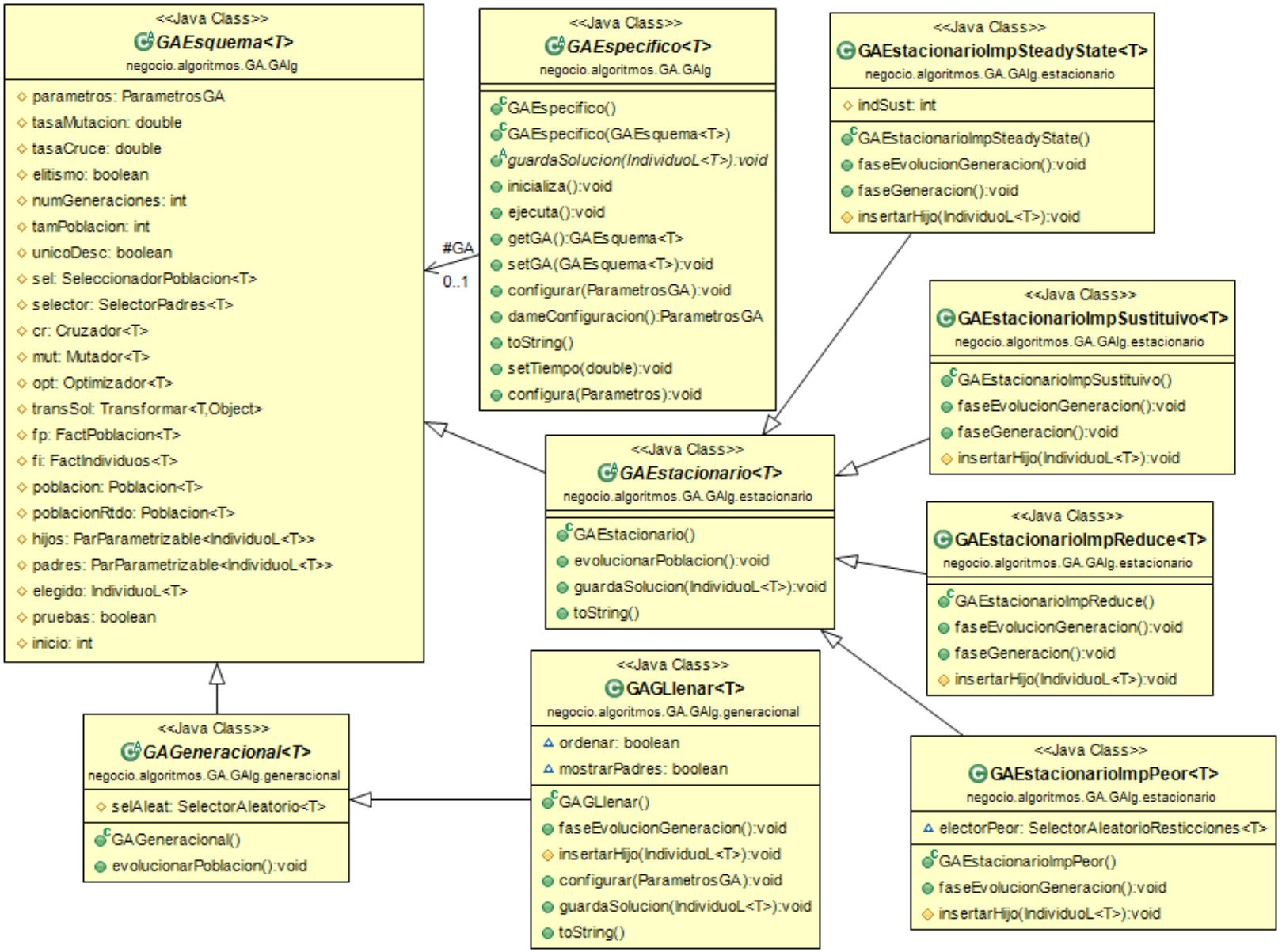

<<Java Class>>
CruzadorDPX<T>
negocio.algoritmos.GA.partes.cruzador.DPX
<<Java Class>>
CruzadorOperador<T>
negocio.algoritmos.GA.partes.cruzador.operador
#op
0..1
<<Java Interface>>
Operador<T>
negocio.algoritmos.GA.partes.cruzador.operador.operadores
op(T,T)
<<Java Class>>
CruzadorOX<T>
negocio.algoritmos.GA.partes.cruzador.permutaciones
<<Java Class>>
Cruzador<T>
negocio.algoritmos.GA.partes.cruzador
#cr
0..1
<<Java Class>>
CruzadorUniforme<T>
negocio.algoritmos.GA.partes.cruzador.UPX
<<Java Class>>
CruzadorSPX<T>
negocio.algoritmos.GA.partes.cruzador.SPX
<<Java Class>>
CruzadorTasa<T>
negocio.algoritmos.GA.partes.cruzador
<<Java Class>>
CruzadorIntMediaAritm
negocio.algoritmos.GA.partes.cruzador.numericos
<<Java Class>>
CruzadorOperadorBooleano
negocio.algoritmos.GA.partes.cruzador.operador


<<Java Interface>>
Mutador<T>
negocio.algoritmos.GA.partes.mutador
mutar(IndividuoL<T>):void
setTasaMut(double):void
#mutador
0..1
<<Java Class>>
MutadorBinarioNegadorTotal
negocio.algoritmos.GA.partes.mutador.basicos
tasaMutacion: double
MutadorBinarioNegadorTotal()
mutar(IndividuoL<Boolean>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorInsercion<T>
negocio.algoritmos.GA.partes.mutador.permutacion
tasaMutacion: double
r: Random
MutadorInsercion()
mutar(IndividuoL<T>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorBinario
negocio.algoritmos.GA.partes.mutador.basicos
rnd: Random
MutadorBinario()
mutar(IndividuoL<Boolean>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorTasa<T>
negocio.algoritmos.GA.partes.mutador
tasaMutacion: double
r: Random
MutadorTasa(Mutador<T>)
mutar(IndividuoL<T>):void
setTasaMut(double):void
getTasaMut():double
toString()
<<Java Class>>
MutadorDosOptimo
negocio.algoritmos.GA.partes.mutador.basicos
optimizador: DosOptimo
tasaMutacion: double
r: Random
mapa: Mapa
MutadorDosOptimo(Mapa)
mutar(IndividuoL<Integer>):void
setTasaMut(double):void
<<Java Class>>
MutadorIntercambio<T>
negocio.algoritmos.GA.partes.mutador.permutacion
tasaMutacion: double
r: Random
MutadorIntercambio()
mutar(IndividuoL<T>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorInversion<T>
negocio.algoritmos.GA.partes.mutador.permutacion
tasaMutacion: double
r: Random
MutadorInversion()
mutar(IndividuoL<T>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorMezcla<T>
negocio.algoritmos.GA.partes.mutador.permutacion
tasaMutacion: double
r: Random
MutadorMezcla()
mutar(IndividuoL<T>):void
setTasaMut(double):void
toString()
<<Java Class>>
MutadorBinarioMult
negocio.algoritmos.GA.partes.mutador.basicos
rnd: Random
tasaMutacion: double
MutadorBinarioMult()
mutar(IndividuoL<Boolean>):void
setTasaMut(double):void
toString()


<<Java Interface>>
PoblacionEstacionariaReduccion<T>
negocio.algoritmos.GA.poblacion.poblaciones
reduce():void
<<Java Interface>>
PoblacionEstacionaria<T>
negocio.algoritmos.GA.poblacion.poblaciones
borraIndividuo(int):void
<<Java Interface>>
PoblacionEstacionariaSustitutiva<T>
negocio.algoritmos.GA.poblacion.poblaciones
sustituyePor(IndividuoL<T>):void
<<Java Interface>>
Poblacion<T>
negocio.algoritmos.GA.poblacion.poblaciones
guardarIndividuo(IndividuoL<T>):void
dameMejorIndividuo():IndividuoL<T>
size():int
capacidad():int
setCapacidad(int):void
estaVacia():boolean
estaLLena():boolean
getIndividuo(int):IndividuoL<T>
estaOrdenada():boolean
esOrdenable():boolean
ordenar():void


<<Java Interface>>
Poblacion<T>
negocio.algoritmos.GA.poblacion.poblaciones
guardarIndividuo(IndividuoL<T>):void
dameMejorIndividuo():IndividuoL<T>
size():int
capacidad():int
setCapacidad(int):void
estaVacia():boolean
estaLLena():boolean
getIndividuo(int):IndividuoL<T>
estaOrdenada():boolean
esOrdenable():boolean
ordenar():void
<<Java Class>>
IndividuoL<T>
negocio.algoritmos.GA.poblacion.individuos
genes: List<T>
cambiado: boolean
mostrarCadaGen: boolean
defecto: T
getFitness():double
clonar():IndividuoL<T>
compareTo(IndividuoL<T>):int
getGenes():List<T>
setGenes(List<T>):void
esVacio():boolean
size():int
getGen(int)
setGen(int,T):void
iterator():Iterator<T>
contains(T):boolean
esFactible():boolean
IndividuoL()
IndividuoL(int)
IndividuoL(int,T)
llenarConDefecto(int):void
toString():String
isMostrarCadaGen():boolean
setMostrarCadaGen(boolean):void


<<Java Interface>>
Poblacion<T>
negocio.algoritmos.GA.poblacion.poblaciones
guardarIndividuo(IndividuoL<T>):void
dameMejorIndividuo():IndividuoL<T>
size():int
capacidad():int
setCapacidad(int):void
estaVacia():boolean
estaLLena():boolean
getIndividuo(int):IndividuoL<T>
estaOrdenada():boolean
esOrdenable():boolean
ordenar():void
#p
0..1
<<Java Class>>
IndividuoL<T>
negocio.algoritmos.GA.poblacion.individuos
genes: List<T>
cambiado: boolean
mostrarCadaGen: boolean
defecto: T
getFitness():double
clonar():IndividuoL<T>
compareTo(IndividuoL<T>):int
getGenes():List<T>
setGenes(List<T>):void
esVacio():boolean
size():int
getGen(int)
setGen(int,T):void
iterator():Iterator<T>
contains(T):boolean
esFactible():boolean
IndividuoL()
IndividuoL(int)
IndividuoL(int,T)
llenarConDefecto(int):void
toString()
isMostrarCadaGen():boolean
setMostrarCadaGen(boolean):void
<<Java Class>>
FactPoblacion<T>
negocio.algoritmos.GA.poblacion.fact
creaIndividuo():IndividuoL<T>
damePoblacion(int,boolean):Poblacion<T>
FactPoblacion(FactIndividuos<T>)
FactPoblacion()
llenar(int):void
#fi
0..1
<<Java Class>>
FactIndividuos<T>
negocio.algoritmos.GA.poblacion.individuos.fact
FactIndividuos()
dameIndividuo():IndividuoL<T>


<<Java Class>>
GAEsquema<T>
negocio.algoritmos.GA.GAlg
parametros: ParametrosGA
tasaMutacion: double
tasaCruce: double
elitismo: boolean
numGeneraciones: int
tamPoblacion: int
unicoDesc: boolean
sel: SeleccionadorPoblacion<T>
selector: SelectorPadres<T>
cr: Cruzador<T>
mut: Mutador<T>
opt: Optimizador<T>
transSol: Transformar<T,Object>
fp: FactPoblacion<T>
fi: FactIndividuos<T>
poblacion: Poblacion<T>
poblacionRtdo: Poblacion<T>
hijos: ParParametrizable<IndividuoL<T>>
padres: ParParametrizable<IndividuoL<T>>
elegido: IndividuoL<T>
pruebas: boolean
inicio: int
<<Java Class>>
GAEspecifico<T>
negocio.algoritmos.GA.GAlg
GAEspecifico()
GAEspecifico(GAEsquema<T>)
guardaSolucion(IndividuoL<T>):void
inicializa():void
ejecuta():void
getGA():GAEsquema<T>
setGA(GAEsquema<T>):void
configurar(ParametrosGA):void
dameConfiguracion():ParametrosGA
toString()
setTiempo(double):void
configura(Parametros):void
#GA
0..1
<<Java Class>>
GAEstacionarioImpSteadyState<T>
negocio.algoritmos.GA.GAlg.estacionario
indSust: int
GAEstacionarioImpSteadyState()
faseEvolucionGeneracion():void
faseGeneracion():void
insertarHijo(IndividuoL<T>):void
<<Java Class>>
GAEstacionarioImpSustituivo<T>
negocio.algoritmos.GA.GAlg.estacionario
GAEstacionarioImpSustituivo()
faseEvolucionGeneracion():void
faseGeneracion():void
insertarHijo(IndividuoL<T>):void
<<Java Class>>
GAEstacionario<T>
negocio.algoritmos.GA.GAlg.estacionario
GAEstacionario()
evolucionarPoblacion():void
guardaSolucion(IndividuoL<T>):void
toString()
<<Java Class>>
GAEstacionarioImpReduce<T>
negocio.algoritmos.GA.GAlg.estacionario
GAEstacionarioImpReduce()
faseEvolucionGeneracion():void
faseGeneracion():void
insertarHijo(IndividuoL<T>):void
<<Java Class>>
GAGeneracional<T>
negocio.algoritmos.GA.GAlg.generacional
selAleat: SelectorAleatorio<T>
GAGeneracional()
evolucionarPoblacion():void
<<Java Class>>
GAGLlenar<T>
negocio.algoritmos.GA.GAlg.generacional
ordenar: boolean
mostrarPadres: boolean
GAGLlenar()
faseEvolucionGeneracion():void
insertarHijo(IndividuoL<T>):void
configurar(ParametrosGA):void
guardaSolucion(IndividuoL<T>):void
toString()
<<Java Class>>
GAEstacionarioImpPeor<T>
negocio.algoritmos.GA.GAlg.estacionario
electorPeor: SelectorAleatorioResticciones<T>
GAEstacionarioImpPeor()
faseEvolucionGeneracion():void
insertarHijo(IndividuoL<T>):void

El centro del *framework* es una clase abstracta *GAEsquema<T>*, la cual está compuesta por las todas las partes que componen un algoritmo genético: selectores, cruzadores, mutadores, … Además, encapsulamos los parámetros de los algoritmos en una clase *ParametrosGA*. Para crear la población inicial, encapsulamos unas factorías para la población y para los individuos.

Siguiendo inicialmente el patrón de construcción *Builder*, hemos creado una estructura para crear un algoritmo genético. Es necesario crear un builder, *BuilderGA_Esp* que herede de la clase *builderGA* en el que se indique qué partes vamos a meter, es decir, qué selector, mutador, cruzador, ... Además, para crear un algoritmo genético para un problema específico hay que crear otro *builder* concreto.

Los selectores seleccionan un individuo de una población. Para ello, se pasa por parámetro la población de la que elegir un individuo. Los tipos de selectores son los que aparecen en el correspondiente diagrama. Como ya dijimos, los más comunes son los selectores de ruleta, torneo y ranking.

Los cruzadores, dados dos individuos, devuelven la descendencia de estos. Hemos implementado cruzadores básicos como SPX (Single Point Crossover), DPX (Double Point Crossover), UPX (Uniform Point Crossover), más complejos como CX (Cycle Crossover), PMX (Partially Mapped Crossover), EX (Edge Crossover).

Los mutadores modifican alguno o varios genes de un individuo. Como ya se explicó, existen varias formas de realizar mutaciones. Hemos implementado dichas formas de mutar según las clases e interfaces del correspondiente diagrama.

La estructura de una población contiene a los individuos de una generación.
Esta estructura está especificada dentro de la interfaz *Población*, la cual especifica los métodos que deben implementar todos los tipos de poblaciones. Entre estos tenemos, métodos para insertar individuos, para saber cuál mejor, para comprobar si la población está vacía o está llena, etc.

Según el tipo de algoritmo genético, tendremos que usar un tipo de población T distinto. Además, como podemos ver, una población está compuesta por individuos.

Un individuo lo hemos implementado como una lista de genes. Aunque no aparezca en este diagrama, existe una interfaz que engloba métodos para obtener el fitness de un individuo, para comparar individuos, para insertar genes, para obtener genes, para clonar individuos, etc.

La clase que implementa a los individuos es una clase parametrizada con un tipo T, ya que existen diversos tipos de individuos, entre los cuales podemos encontrar de tipo booleano, entero, …

Las factorías las implementamos para la creación de los elementos fundamentales de los algoritmos genéticos. Tenemos:

- *Factoría de individuos*: crea soluciones (genéricas o específicas del problema) encapsuladas en un individuo de tipo T.

- *Factoría de población*: crea una población para individuos de tipo T. Puede crearse llena de individuos o vacía. Incluye una factoría individual, y se usa para crear una población llena.

Como ya se sabe, existen varios tipos de algoritmos genéticos, los cuales han sido especificados e implementados según el correspondiente diagrama.

Bibliografía

[1] Ricardo Peña Mari, Marco Antonio Gómez Martín, Pedro Antonio González Calero, Manuel Freire, Ramón González del Campo, Gonzalo Méndez, Antonio Sanchez, Miguel Valero, Estructuras de Datos y Algoritmos - APUNTES DE CLASE, Madrid: Facultad de Informatica - Universidad Complutense de Madrid, 1 de octubre de 2012.

[2] Wikipedia, the free encyclopedia, «NP (complexity)» octubre 2015. [En línea]. Available: https://en.wikipedia.org/wiki/NP_(complexity).

[3] Wikipedia, the free encyclopedia, «P (complexity)» [En línea]. Available: https://en.wikipedia.org/wiki/P_(complexity).

[4] A. Jiménez, «Xataka ciencia» 26 Agosto 2006. [En línea]. Available: http://www.xatakaciencia.com/matematicas/p-versus-np-nunca-lo-entendiste.

[5] E. Besada-Portas, A. Herrán« OPTIMIZACIÓN HEURÍSTICA Y APLICACIONES - MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS Y DE CONTROL» Universidad Complutense de Madrid, UNED, Madrid.

[6] Wikipedia, the free encyclopedia, «Problema del viajante» [En línea]. Available: https://es.wikipedia.org/wiki/Problema_del_viajante.

[7] Wikipedia, the free encyclopedia, «Problema de la mochila» [En línea]. Available: https://es.wikipedia.org/wiki/Problema_de_la_mochila.

[8] N. Martí Oliet, Y. Ortega Mallén, A. Verdejo, Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos, capítulo 15, Garceta, 2013.

[9] E. Pérez, «Guía para recién llegados a los Algoritmos genéticos».

[10] Abdelmalik Moujahid, Iñaki Inza, Pedro Larrañaga «Tema 2. Algoritmos Genéticos» Departamento de Ciencias de la Computación e Inteligencia Artificial Universidad del País Vasco–Euskal Herriko Unibertsitatea.

[11] Curso de posgrado, UC3M, «Algoritmos evolutivos y memeticos» Madrid.

[12] L. Recalde, «Esquemas algorítmicos - Algoritmos genéticos».

[13] Javier Hidalgo Sánchez, Jesús Ignasio Turrado Martínez «Algoritmos genéticos: Aplicación al problema de la mochila» Universidad Carlos III de Madrid, Madrid.

[14] Marco Dorigo, Gianni Di caro «The Ant Colony Optimization Meta-Heuristic» Université Libre de Bruxelles.

[15] Wikipedia, the free encyclopedia, «2-opt» [En línea]. Available: https://en.wikipedia.org/wiki/2-opt.

[16] Java, «Java y el explorador Google Chrome» [En línea]. Available: https://www.java.com/es/download/faq/chrome.xml.

Un libro abierto es un cerebro que habla; cerrado, un amigo que espera; olvidado, un alma que perdona; destruido, un corazón que llora.", Mario Vargas Llosa.